Mapper's Guide for

DoD E-Business Exchange System

Version 3.0

June 2000

Inter-National Research Institute, Inc.
12350 Jefferson Avenue, Suite 400
Newport News, Virginia 23602

Mapper's Guide for DEBX Version 3.0

Mapper's Guide

Contents

List of Figures

This page intentionally left blank.

1.0 Document Overview

This document serves two purposes: 1) To give an overview of translation processing within the DoD E-Business Exchange System (DEBX) and 2) To provide instructions on creating a set of translation maps and supporting products for a new user-defined file (UDF) type. This document assumes that you have some previous knowledge of the TSI Mercator® development suite. You should also be familiar with the DEBX software, the X12 messaging structure, and the UDF messaging structure to be translated.

2.0 Referenced Documents

The following documents are referenced in this *Mapper’s Guide*. In the event of a later version of a referenced document being issued, the later version shall supersede the referenced version.

- ASC X12 Electronic Data Interchange X12 Standards, Release 4010, DISA, December 1997.
- Mercator Documentation CD, Version 1.4.2, 1998.

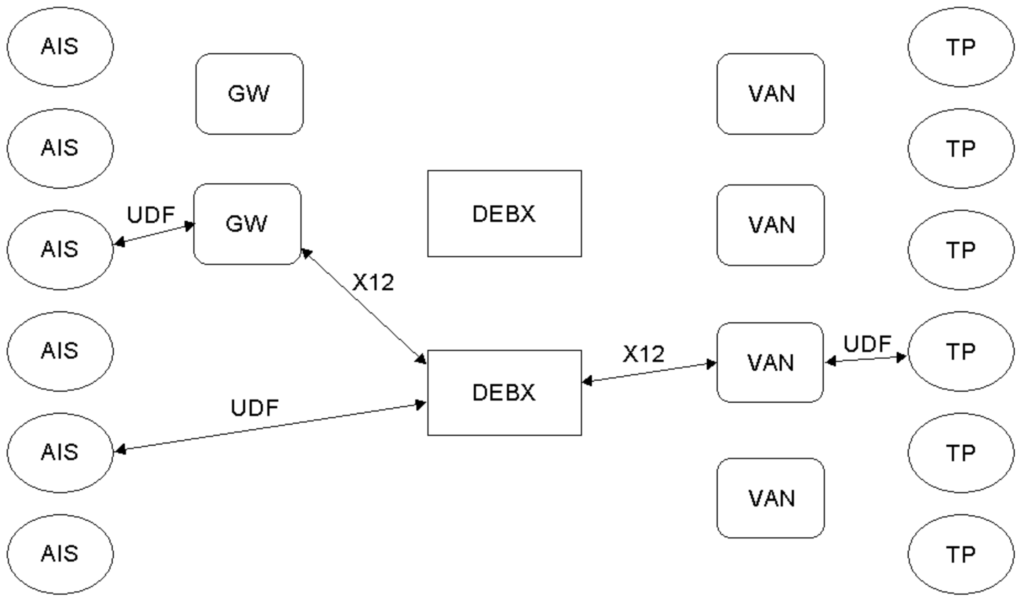
3.0 Translation Overview

The Electronic Commerce Infrastructure (ECI) comprises government and industry systems that conduct business using Electronic Data Interchange (EDI). DEBX facilitates EDI between government sites (i.e., Automated Information Systems [AISs]) and industry sites (i.e., trading partners).

DEBX processes two categories of messages — X12 format and user defined file (UDF) format. The X12 format complies with the American National Standards Institute (ANSI) benchmark for electronic commerce. As their name implies, UDF formats vary, depending on their origination point. When DEBX receives a UDF, it translates the UDF into X12 format as part of processing. After translation, an X12 may be translated again to another UDF format depending upon its destination.

Figure 3-1 depicts a typical message exchange: An AIS sends a UDF message to DEBX, which translates the message to X12. (Some AISs do not send UDFs directly to DEBX. In this case, the AIS sends a UDF to a gateway [GW], which translates the message to X12 and then sends it to DEBX.) DEBX forwards the X12 message to a Value Added Network (VAN). The VAN, in turn, translates the message to a UDF format that a trading partner can accept and sends the UDF to the trading partner. If the trading partner sends back a message in response, that message (a UDF) travels to the VAN, which translates it to X12 format and forwards it to DEBX. When DEBX receives the message, it translates the message to UDF and then sends it to the AIS. (For some AISs, DEBX sends the X12 to a GW, which translates the message to UDF and then sends it to the AIS.)

Figure 3-1 Message Exchange



3.1 Transaction Types

Table 3-1 lists the most commonly used transaction types for EDI.

Table 3-1 Transaction Sets

Identifier	Title
810	Invoice
820	Payment Order/Remittance Advice
821	Financial Information Reporting
824	Application Advice
836	Procurement Notice
838	Trading Partner Profile
840	Request for Quotation
843	Response to Request for Quotation
850	Purchase Order
855	Purchase Order Acknowledgment
860	Purchase Order Change Request – Buyer Initiated
865	Purchase Order Change Acknowledgment/ Request – Seller Initiated
997	Functional Acknowledgment

3.2 Maps

DEBX uses the Mercator software to perform the actual translation of messages using a series of maps. A *map* is a translation specification to Mercator. A collection of maps and processing rules on one type of UDF is called a *map family*. A UDF-to-X12 map handles all UDFs for all transaction types. Similarly, an X12-to-UDF map handles all X12s for all transaction types. For example, if an AIS sends 840, 850, and 860 transaction types, a single UDF-to-X12 map for the map family handles the mapping for all of these transaction types.

DEBX supports two types of translation acknowledgments: 997 and 824. A 997 acknowledgment contains status information about an X12-to-UDF translation, and an 824 acknowledgment contains status information about a UDF-to-X12 translation. A map family that includes acknowledgment maps produces an acknowledgment message for each translation. A map family that does not include acknowledgment maps does not generate acknowledgments. DEBX administrators may configure the interface between DEBX and the message originator to send no acknowledgments, to send acknowledgments for translation failures, or to send acknowledgments for all translations.

The interface between DEBX and an external source is known as a *channel*. For a complete description of communications channels, see the Help system.

Map families are specified per channel; therefore, each message received on a channel is of the specified message type, and each message sent to a channel is translated to that same message type. Each map invoked for a given family is specified via a message description file. Through keywords, the message description file specifies each of the maps associated with that family. This file also contains descriptive information that is shown during channel configuration. For an explanation of the message description file, see Appendix B. Associated with each map family are various map documents that provide details on the Implementation Conventions (IC) used, the UDF specifications, and so on.

A typical map family consists of one or more of the following types of maps:

- Premap 1

This map is used when a two-stage file manipulation is required prior to the regular UDF-to-X12 translation. It typically preprocesses an incoming file to produce a series of contiguous single-addressee messages. This processing occurs for each message type that has a sequence of addresses above a single message body. A Premap1 is used for UDF-to-X12 translation, and it is optional.

- Premap

This map bounds, filters, and pads each message in an input file so that the translator can extract each message for mapping. A Premap is used only in UDF-to-X12 translation, and it is optional. If a Premap1 is used, a Premap must also be used.

- UDF to X12 map

This map accepts as input either a raw UDF message or a message prepared by the premap(s) and produces the corresponding X12 message and audit log. The audit log is used as input to the optional 824gen map to produce a translation status message.

- 824gen map

Using input from the audit log that was produced through the UDF-to-X12 map execution, this map produces an 824 X12 translation status message, known as an acknowledgment. Note that if this acknowledgment is sent back to the UDF originator, this translation status message is translated to an 824 UDF.

- X12 to UDF map

This map takes an X12 message as input and produces the corresponding UDF message and audit log. This audit log is used as input to the optional 997gen map to produce a translation status message.

- 997gen map

Using input from the audit log that was produced through the X12-to-UDF map execution, this map produces a 997 X12 that may be sent to the X12 originator. Note that during outgoing (X12-to-UDF) translation, a 997 acknowledgment is not generated for a message that was received on a UDF channel or for a system-generated message, such as an 824 acknowledgment message created during UDF-to-X12 translation.

Table 3-2 lists the default location of each map family component. Note that the location of individual maps (ported to HP-UX) and optional look-up tables can be overridden using full paths or expandable tokens in the message description file. For a description of map names, see Appendix B.

Table 3-2 Translation Map Default Directories

This directory	Contains
/h/data/global/EC/Messages/Maps/<map_ family>	Map families (individual collections of map files ported to HP-UX); may also contain optional look-up tables
/h/data/global/EC/Messages/MessageDesc /<map_family>	Message description file, which includes descriptions of map files, transaction sets supported, and any unique addressing information
/h/data/local/EC/html/MapDocs/<map_ family>	(Optional) Mapping specifications and implementation conventions, along with the HTML files that serve as their table of contents

For example, for the Standard Army Accounting and Contracting System (SAACONS), the default location for the map files and the look-up table files is /h/data/global/EC/ Messages/Maps/SAACONS; the location for the message description file is /h/data/ global/EC/Messages/MessageDesc/SAACONS; and the location for the HTML file and all of the documents that it references is /h/data/local/EC/html/MapDocs/SAACONS.

3.3 DEBX Translation Programs

The following DEBX programs are essential to translation:

- comms, emaild, email_send, ftpd

These programs are responsible for receiving and sending messages to and from DEBX. If a program is receiving messages, it is called InComms, and if a program is sending messages, it is called OutComms.

- InXlator

The InXlator program is responsible for interacting with Mercator using all of the specified premaps and maps to perform UDF-to-X12 translation and to generate the associated acknowledgment. The InXlator also queries the system setup database and TPDB on behalf of the maps and provides other reference services. (For more information on this query process, see Appendix A.)

- Router

The Router program determines target channels based on how the routing database is configured. If the target channel is a UDF channel, the message is queued to the OutXlator. If the target channel is not a UDF channel, the message is queued directly to OutComms.

- OutXlator

The OutXlator program is responsible for interacting with Mercator using the specified maps to perform X12-to-UDF translation and to generated the associated acknowledgment. The OutXlator also queries the system setup database and TPDB on behalf of the maps and provides other reference services. (For more information on this query process, see Appendix A.)

3.4 UDF-to-X12 Translation Basics

Each UDF message translates into a single X12 transaction. So, for example, if a UDF input file contains five UDFs, five separate X12 transactions result from the UDF-to-X12 translation process. For each X12 transaction produced, the X12 envelope information (as described in this section) is generated by the map. During each map invocation, the map may request database data or specify other values within the translator, as described in Appendix A.

3.4.1 X12 Envelope Information

An *interchange* is the information in a message within one ISA and the corresponding IEA segment, and a *functional group* is the information within one GS and the corresponding GE segment. These components define the standard enveloping of an X12 transaction, and this envelope contains addressing information. Each of the elements listed in Table 3-3 is created by the map to produce an X12 envelope. (For a description of the X12 interchange [ISA/IEA] and functional group [GS/GE] envelopes, see the *ASC X12 Electronic Data Interchange X12 Standards, Release 4010.*)

Table 3-3 X12 Envelope Information

Field number	Field name	Value/Rule
ISA01	Authorization Information Qualifier	“00”
ISA02	Authorization Information	None
ISA03	Security Information Qualifier	“00”
ISA04	Security Information	None
ISA05	Interchange Sender ID Qualifier	See Section 3.4.2
ISA06	Interchange Sender ID	See Section 3.4.2
ISA07	Interchange Receiver ID Qualifier	See Section 3.4.2
ISA08	Interchange Receiver ID	See Section 3.4.2
ISA09	Interchange Date	Map rule: CURRENTDATE ()
ISA10	Interchange Time	Map rule: CURRENTTIME ()
ISA11	Interchange Control Standards ID	“U”
ISA12	Interchange Control Version Number	Map family dependent (e.g., “00305”)

Field number	Field name	Value/Rule
ISA13	Interchange Control Number	See Section 3.4.2.2
ISA14	Acknowledgment Requested	"0"
ISA15	Test Indicator	"P"
IEA01	Number of Included Functional Groups	Map rule: COUNT (FuncGroups)
IEA02	Interchange Control Number	Identical to ISA13
GS01	Functional ID Code	2-letter code, dependent on transaction set
GS02	Application Sender's Code	UDF Sender Code
GS03	Application Receiver's Code	UDF Receiver Code
GS04	Date	Map rule: CURRENTDATE ()
GS05	Time	Map rule: CURRENTTIME ()
GS06	Group Control Number	See Section 3.4.2.2
GS07	Responsible Agency Code	"X"
GS08	Version Release Industry Code	Map family dependent (e.g., "003050")
GE01	Number of Transaction Sets Included	Map rule: COUNT (Transactions)
GE02	Group Control Number	Identical to GS06

3.4.2 Envelope Value Generation

The envelope values (listed in Table 3-3) are filled during X12 message creation using the techniques described in Section 3.4.2.1 to 3.4.2.3.

Each AIS identifies itself and addresses UDF messages to recipients in its own unique way. Sometimes this identification information is contained within the UDF; at other times, it is external information, such as the file name extension. An AIS primarily uses the following codes for addressing:

- CAGE – Commercial and Government Entity
- DODAAC – Department of Defense Activity Address Code
- DUNS – Data Universal Numbering System
- DUNS+4 – DUNS number with a 4-character suffix

3.4.2.1 Values from Message Content

Most of the time, an addressing code in message content is an unqualified value. Heuristics must be applied to deduce whether the value is a CAGE code, a DODAAC, and so on.

Typically, the following rules are used for distinguishing codes:

- If a code is 9 numeric characters, it is a DUNS.
- If a code is 13 numeric characters, it is a DUNS+4.
- If a code is 5 characters (1st and 5th numeric; others alphanumeric, except I and O), it is a CAGE.
- Any other code is a DODAAC.

3.4.2.2 Values from Databases that Support Translation

The following two DEBX databases assist in the translation process:

- System setup database

This database specifies a range for ICNs (ISA13s) and another range for GCNs (GS06s). ICNs and GCNs are incremented by DEBX each time an X12 envelope is generated, that is, whenever any of the following events occur:

– A UDF message is translated to an X12 interchange.
– An 824 X12 translation status message is generated.
– A 997 X12 translation status message is generated.

ICNs and GCNs wrap at the end of the range specified in this database.
The system setup database also specifies the ISA05/ISA06 that is used in the X12 envelope for the translated message.

- Trading partner database

Each trading partner registers with the Central Contractor Registry (CCR) via an 838 message that contains information on the trading partner. After verifying this information, CCR forwards the 838 to DEBX. In turn, DEBX parses the 838 and populates the trading partner database (TPDB) with information such as the name and address of the trading partner, remittance information, points of contact, and CAGE/DODAAC/DUNS for the trading partner. The key field is DUNS.

When an envelope is generated for some UDF families, the TPDB is queried to generate addressing information. (For more information on this query process, see Appendix A.)

3.4.2.3 Values from Look-up Tables

When defined in the message description file, an additional input file called a *look-up table* is passed to the UDF-to-X12 map. Look-up tables provide map family or channel-specific data so that message content or attributes (e.g., received file name extension) may be used to derive an envelope address (i.e., ISA06, ISA08, GS02, or GS03). A look-up table may be edited through the DEBX Edit Channel dialog box, as described in the Help system. For a complete description on path/file name resolution for look-up tables, see Appendix B.

3.5 X12-to-UDF Translation Basics

Each message that is routed to a UDF channel type must be translated to that channel's UDF format prior to transmittal. This principle applies to messages received on an X12 channel, messages received on a UDF channel and translated into the intermediate format (X12), and DEBX-generated X12 824 application advice messages defining the translation status of a received UDF message. During each map invocation, the map may request database data or specify other values within the translator, as described in Appendix A.

By default, if a 997 acknowledgment map is provided, each X12-to-UDF translation generates an X12 997 Functional Acknowledgment message. This acknowledgment message reports on the translation success or failure of an X12 functional group and is generated by parsing the audit log, which is an artifact of the X12-to-UDF translation. The default amount of data that is passed to the X12-to-UDF map is a single functional group (at a time). This default can be overridden by setting the translation level (values ISA, GS, or ST) in the message description file. For a complete description of the message description file, see Appendix B.

3.6 Processing Flow

The following subsections describe the processing flow for UDF-to-X12 translation (Section 3.6.1) and X12-to-UDF translation (Section 3.6.2).

3.6.1 UDF to X12

A channel is configured for the UDF type for which it receives messages. The InComms process for that channel places a received file on a queue for the InXlator. If a Premap1 is listed in the message description file, the InXlator invokes the Premap1 to reorganize the input file into contiguous single-addressee messages. InXlator then invokes the premap (if specified in the message description file). The execution of this premap produces the following files:

- A modified input file that may include filtered values, replaced values, record padding, and/or record initiation tokens.
- An index file specifying the bounds for individual messages in the input UDF file and the modified input file. Note that if a Premap1 is invoked prior to the premap, the output of Premap1 is the input UDF file for premap.

If premap(s) are specified and a premap fails, a default 824 acknowledgment is generated. The 824 acknowledgment may be sent back to the UDF originator if the channel is configured to do so. This default message provides very little detail and notifies the message originator of the translation failure.

If premap(s) are specified and the premap stage is successful, the InXlator uses the index file to divide the input and output files of the premap into single transaction pieces. The premap output pieces are passed individually through the UDF-to-X12 map. If no premap was specified in the message description file, the original received UDF file is passed to the UDF-to-X12 map.

If an 824gen map is provided for the map family, the InXlator executes it after each execution of the UDF-to-X12 map to generate 824 X12 translation status messages.
The InXlator then collects all of the output X12s, collects the individual UDF messages from the input file, extracts the 824 X12s, and queues them to the Router process.

The Router process, using information passed by the InXlator, creates message objects for the X12s and the 824 X12s, linking them appropriately. The Router then routes the X12s to the destination — the X12 to UDF translator (OutXlator) if the destination is a UDF channel or OutComms if the destination is an X12 channel.

If the DEBX administrator has elected to send acknowledgments back to the original channel, and if the acknowledgment messages are generated for the map family, the Router routes the 824 X12s to the OutXlator, which translates them to UDF before sending them to the originator.

3.6.2 X12 to UDF

Using the information passed from the Router, the OutXlator invokes the X12-to-UDF map for translating the X12 to a UDF. Next, if a 997gen translation map is provided for the map family, the OutXlator invokes the 997gen map to produce 997 X12 translation status messages. This invocation of the 997gen map is skipped if the X12 message being translated is an 824 X12 translation status message or if it is just an intermediary message in UDF-to-UDF translation.

The OutXlator then queues the translated message (the UDF) to OutComms and queues the 997 X12s to the Router (to be sent back to the X12 originator if the source channel is configured this way).

If the destination channel for the UDF is of the same type as that of the UDF sender, no invocations of the X12-to-UDF map are made, and the UDF is extracted from the message source and sent to the destination. For example, this capability is used to forward (successfully translated) SAACONS UDFs to the Defense Accounting and Printing System (DAPS).

3.7 Admin Message Processing

DEBX provides an ADMIN tab in the Edit Channel dialog box through which the user can configure how and where to send the 824 and 997 translation status messages (admin messages) on each channel. These messages may be sent back on the incoming channel. (For example, if UDF messages are received via FTP, these admin messages may be sent back via FTP.) Alternately, the translation status messages may be sent as email messages to the email address(es) specified. In addition, the DEBX administrator may specify whether to send all acknowledgments, send only the negative acknowledgments, or not send any acknowledgments.

3.8 Translation Toolbox

The translation process is transparent to DEBX administrators. However, the DEBX administrators may use the translation toolbox interface as a window into the various map invocations that the InXlator and OutXlator perform. The translation toolbox is available from the message log and error queue and allows administrators to view all of the intermediate data such as the premap.out file, the modified input file, and so on. For information on the translation toolbox, see the Help system.

4.0 Type Tree Construction

In order to begin the mapping process, you must either create or use existing type trees to represent the input and output data involved in the translation process for your specific system. Because these type trees will be used in mapping either UDF transactions to X12 transactions or X12 transactions to UDF transactions you must create both UDF and X12 type trees.

4.1 Document Overview

Various types of documents are available to assist you in the map development process. This section provides an overview of these documents.

4.1.1 ANSI ASC X12 Document

The *ASC X12 Electronic Data Interchange X12 Standards* is the standard document for the entire X12 format of messages and is published (in print only) for each version of the X12. For example, version 3010, version 3040, version 3050, and version 4010 each has its own X12 specification.

Each transaction type that can be used in an X12 EDI exchange is described in this document. This document also describes each segment that is part of a transaction and lists the data dictionary for each element of each segment.

4.1.2 Implementation Conventions (IC)

Implementation Conventions are also sometimes called Implementation Guidelines (IG). The X12 standard is an all-encompassing document that serves everyone who exchanges messages via EDI. However, each industry pares down the standard to suit its particular needs and creates an industry-specific IC.

The Government issues this pared down standard that suits its requirements. Some of the specifics of this paring down are as follows:

- Some optional segments in the X12 standard might be marked as mandatory in the IC.
- Other optional segments in the X12 standard might be marked as not used in the IC.
- Each element might be restricted to having only a few values taken from the entire data dictionary.

For each version of X12 that ANSI specifies, there is a corresponding set of ICs.

4.1.3 Part 10

Part 10 deals primarily with X12 enveloping and the acknowledgment model that is to be used for EDI in the Government. This document does *not* specify an acknowledgment model for UDF-to-X12 translations. It does, however, specify Point-of-translation (POT) to Point-of-translation acknowledgments for X12s.

4.1.4 UDF Specifications

Each agency that sends UDFs to DEBX, documents the format of UDF data that it sends.

4.2 Creating X12 Type Trees

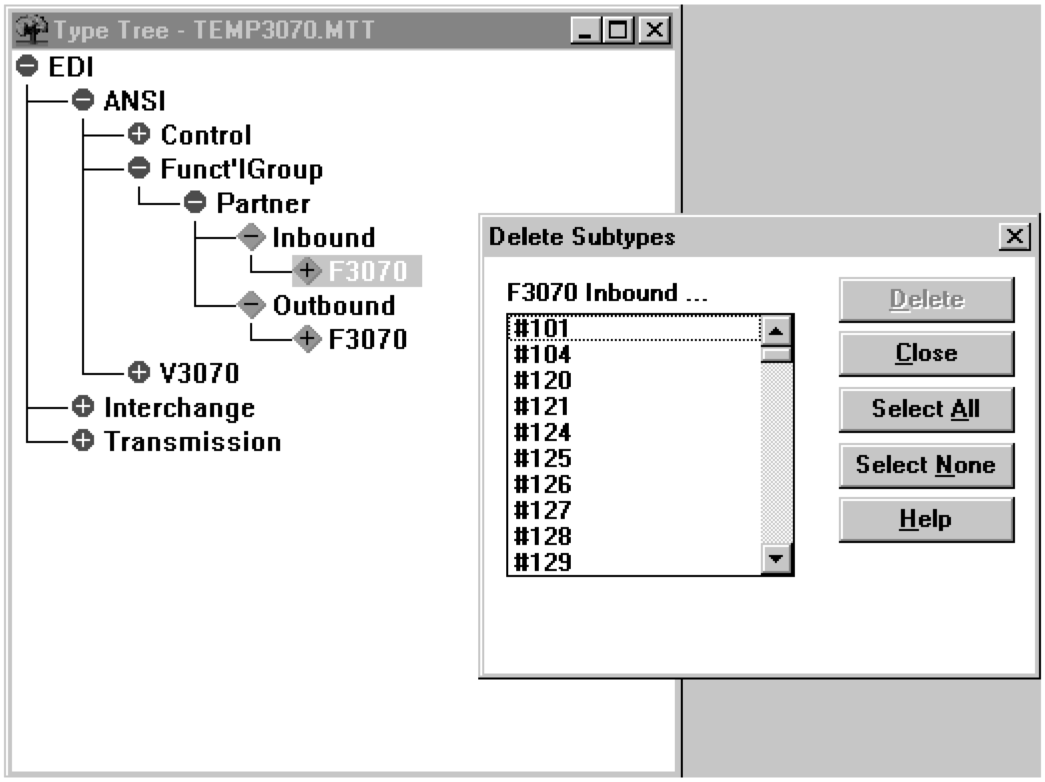
You may create a type tree by paring down an existing ANSI X12 type tree. You should do this only when the X12 specification is taken directly from the ANSI X12 standards.

To make a copy of the standard EDI type tree

The first step in creating an X12 type tree is to make a copy of the standard EDI type tree, containing only the transactions you specify (also known as paring down a standard type tree).

1. Open the EDI type tree (e.g., ANSI3070.MTT, which contains the full ANSI 3070 version release). When creating a multi-version tree, start with the latest version release that you will be using.
2. To save the type tree as your new IC type tree file and rename the file, select File > Save As.
3. Under Inbound Partner Funct'lGroup ANSI EDI, select F3070 (or Fnnnn, where nnnn is the 4-digit version release indicator code).
4. Select Type > Delete Subtypes. The Delete Subtypes dialog box (Figure 4-1) appears, listing all of the F3070 Inbound Partner Funct'lGroup objects contained within the type tree.

Figure 4-1 Delete Subtypes Dialog Box



5. In the Delete Subtypes dialog box, click Select All. All of the subtypes in the list are highlighted.
6. Deselect each group *to keep* by clicking it, which removes the highlighting. Some transactions are included as part of a group of related elements. For example, transaction 821 is included with the 827 transaction as part of the FR group.
7. To remove the groups, click Delete.
8. Repeat *Steps 3* to *7* for F3070 under Outbound Partner Funct'lGroup ANSI EDI to remove the Outbound Partner Groups that are not needed.
9. Save the IC file.

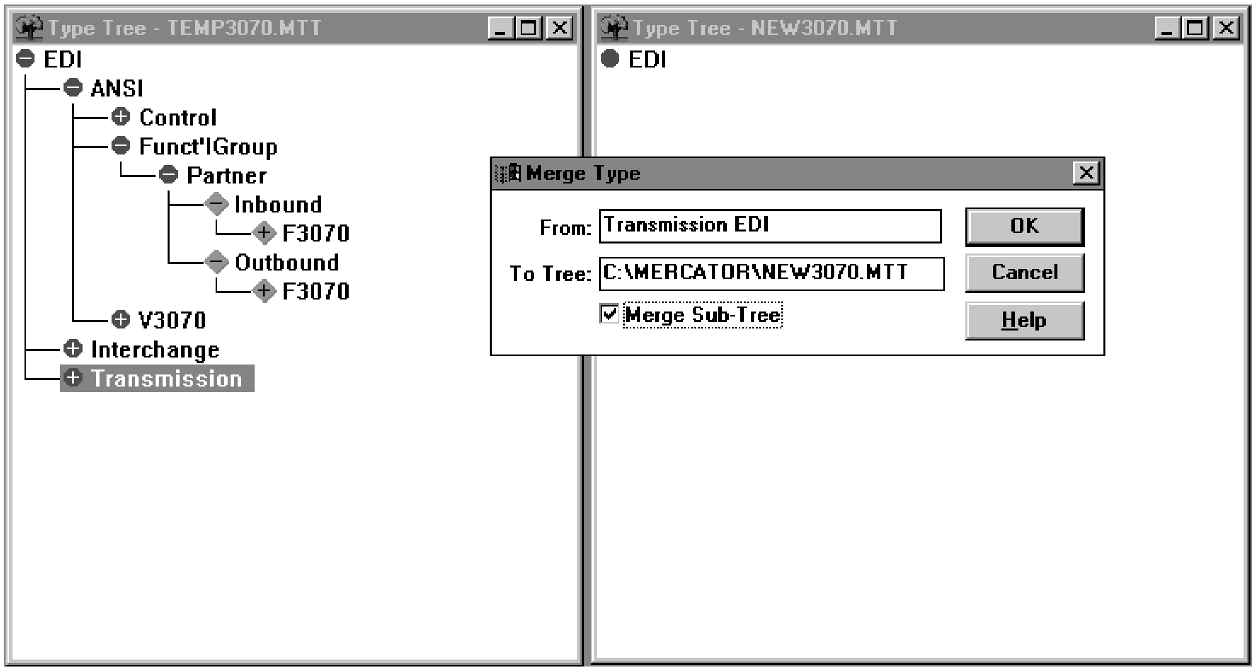
At this point, many unwanted subtypes remain in the type tree. You may remedy this situation by exercising one or both of the following two options. One option is to remove an unnecessary directional transaction set. Another option is to remove unnecessary transactions within a group containing desired transactions. To prune the type tree to the desired scope, perform the steps in this section again.

To merge an IC type tree into a new tree

This process removes all unnecessary data elements that are not referenced in a pared-down type tree.

1. Create a new type tree, using EDI as the root name. Save the new tree to a different name than that used in *Step 2* of *To make a copy of the standard EDI type tree*.
2. From your saved IC file (created in *Step 2* of *To make a copy of the standard EDI type tree*), select the type Transmission EDI.
3. Select Type > Merge. The Merge Type dialog box (Figure 4-2) appears.
4. Select the type tree created in *Step 1*, and then select the Merge Sub-Tree check box.

Figure 4-2 Merge Type Dialog Box



From: Transmission EDI

To Tree: C:\MERCATOR\NEW3070.MTT

☑ Merge Sub-Tree

5. Click OK.
6. After the merge finishes, save the new type tree.

To clean up unused elements

Although you may have removed some elements in the merging process, other unnecessary elements remain. For example, in the case of removing a directional set of transactions (e.g., Outbound) presented in *To make a copy of the standard EDI type tree*, the Funct'lGroup Partner is removed, but the Transmission and Interchange remain. These directional elements can be removed by selecting Transmission or Interchange and selecting Type > Delete Subtypes (For this example, selecting Outbound and then clicking Delete.)

The same process may also be performed by selecting the part to be removed (Outbound Transmission or Outbound Interchange) and pressing the [Delete] key on the keyboard. A dialog box appears confirming that you wish to delete that type.

NOTE: If you wish to later add a different version transaction in a certain direction, do *not* remove the directional transaction set. Perform the process detailed in *To make a copy of the standard EDI type tree,* but delete *all* transaction sets for that direction.

Another commonly unused set of elements which remain are those elements related to UCS transactions, as opposed to X12 transactions. UCS types appear in the following locations:

- Inbound/Outbound Transmission EDI
- Inbound/Outbound Interchange EDI
- Variant Control ANSI EDI

The first two types can be removed using the Delete Subtypes method (described in *To make a copy of the standard EDI type tree*), by selecting Inbound or Outbound within the Interchange EDI or Transmission EDI levels. For the Variant Control type, use the same method, selecting Control ANSI EDI.

Another way to delete the types is to select the location to be removed (UCS Inbound/Outbound Transmission EDI, UCS Inbound/Outbound Interchange EDI, and Variant Control ANSI EDI) and press the [Delete] key on the keyboard. A dialog box appears confirming that you wish to delete that type.

To analyze a new type tree

This procedure ensures your new tree is valid.

1. Select Tree > Analyze, or select the analyze tool from the toolbar (a check mark).
2. Select either Logic or Both, and then click OK.
3. Correct any errors that result.

NOTE: If you wish to later add a different version transaction and have deleted an Inbound or Outbound Funct'lGroup, you will not be able to correct the error that will result. Once the different version transaction is added in the direction that was deleted, and the tree is analyzed, the errors should be resolved.

4. Save the type tree.

To add additional functional groups to a new IC type tree

Note that this process may not be necessary.

1. Open the new IC type tree.
2. Open the standard EDI type tree.
3. Under Inbound/Outbound Partner Funct'lGroup ANSI EDI, select the functional group that contains the transaction set to be added.
4. Select Type > Merge. The Merge Type dialog box (Figure 4-2) appears.
5. Select the IC type tree, and then select the Merge Sub-Tree check box.
6. Click OK.
7. After the Merge finishes, save the industry subset file.

To analyze a new type tree

This procedure ensures your new tree is valid.

1. Select Tree > Analyze, or select the analyze tool from the toolbar (a check mark).
2. Select either Logic or Both and then click OK.
3. Correct any errors that result.
4. Save the type tree.

4.3 Creating UDF Type Trees

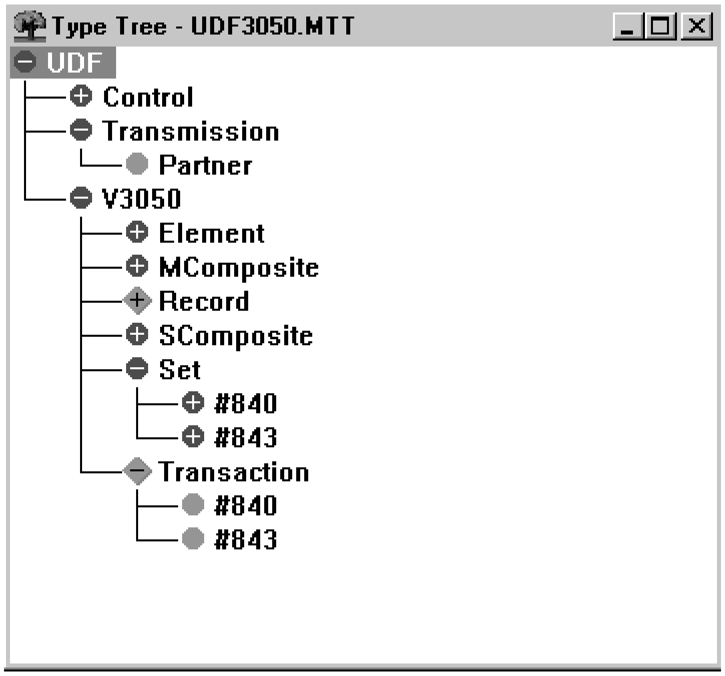
If the data within a UDF is similar in representation to that of an ANSI X12 structure, you may use a type tree of the corresponding X12 transaction set, and derive the UDF data structure from a subset of this tree. The record format and UDF enveloping may be changed as necessary to match the UDF specification.

It may be necessary to create the type tree from scratch. If this is the case, you may find it helpful to look at an existing type tree to learn what your type tree should look like.

Each UDF varies depending upon the logical map requirements; however, a consistent enveloping scheme should be used throughout all UDF type trees. Consistency makes it easier to understand a type tree, and makes the process of merging type trees easier.

For consistency, the root node of the UDF type tree should be named *UDF*. This node denotes the type of message that is contained in the type tree. The following categories should be under the UDF root node: Control, Transmission, and any version or system-specific transaction identifiers (e.g. V3050, V3070). The Control category should contain all delimiter information and any structure used to contain only X12 enveloping information (ISA and GS information). In the example displayed in Figure 4-3, only version-specific identifiers are used. While a UDF does not necessarily have version identifiers, it is sometimes useful to describe differing UDF transactions by the version and transaction type of the corresponding X12 to which it will be translated.

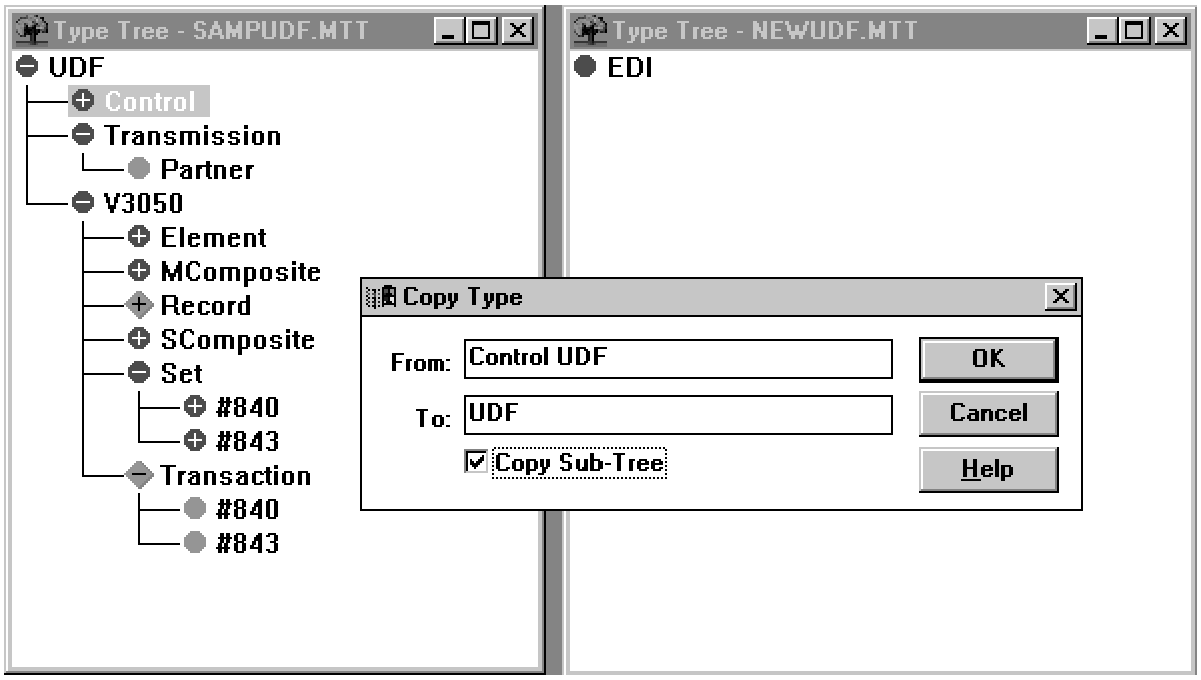
Figure 4-3 UDF Enveloping Scheme Type Tree Window



Because the data represented in the UDF appears in the form of a continuous stream of bytes, a method must be used to determine the separation between individual Elements, and, at a higher level of abstraction, Records, Loops, and Transactions. To help in this separation process, you may use Items known as Control Delimiters. In many cases, the following three delimiters are used: the Composite, Element, and Terminator delimiters. The most commonly used delimiter, the Terminator delimiter signifies the end of a record, usually in the form of a carriage return/line feed sequence or some other end-of-line signal. The Composite delimiter marks the boundaries between Elements contained within a Composite. The Element delimiter marks the boundaries between Elements and Composites used within a record. The Composite and Element Delimiters are mainly used with data, which contains delimited records, as opposed to fixed-length records.

The Control category may be created using the new element tool (leaf tool), or it may be copied from an existing type tree. To copy the category, select Control on the original type tree and then select Type > Copy. The Copy Type dialog box (Figure 4-4) appears. To copy the Control category, click on the destination type tree, select the Copy Sub-Tree check box, and then click OK.

Figure 4-4 Copy Type Tree Dialog Box



From: Control UDF

To: UDF

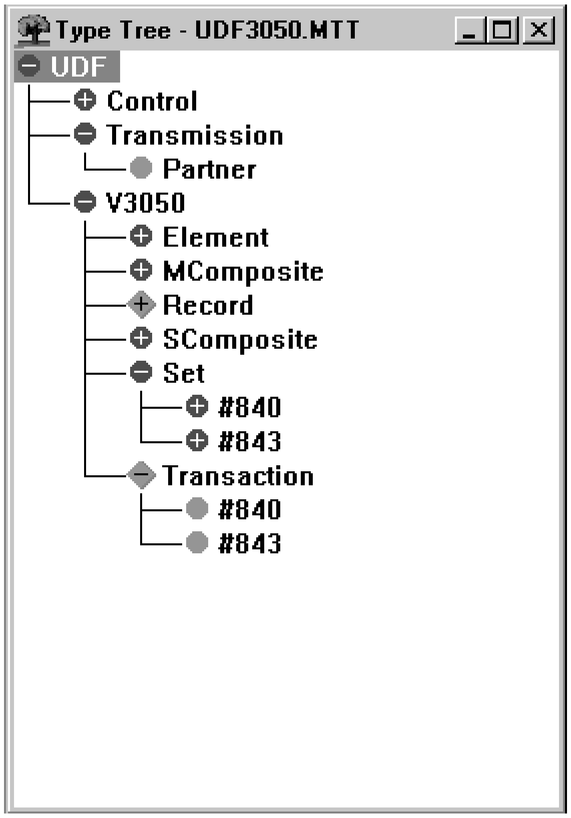
[✓] Copy Sub-Tree

When using the copy method, you may find that some of the delimiters are unnecessary. These delimiters or elements may be removed by selecting each one and then pressing the [Delete] key on the keyboard. A dialog box appears asking you to confirm the deletion. You may also wish to limit the possible values that a specific delimiter can contain. You may view all of the possible values that a delimiter contains by double-clicking on the specific delimiter (e.g., the Terminator delimiter under Delimiter Control UDF). Once you are viewing the list of values, values may be added by clicking on the bottom-most empty square in the Restriction column and entering the new value. Make sure that the correct method of representation is chosen, Hex or Symbol, before entering the value. You may also wish to enter a description of the entered value. If you wish to remove one or more values from the list, you may do this by selecting the value to remove, and then selecting Restriction > Delete. Once all changes have been made, close the restriction window. If changes were made, you are prompted to save the changes.

The basic concept of a file is abstracted as the Transmission level of a type tree (Partner group in Transmission category). A file containing one or more UDFs is, in turn, thought of as a Transmission consisting of one or more Transactions, where a UDF is abstracted as the Transaction level of a type tree (Transaction Partition Group in SysName or V3070, V3050 Category). The Transaction Group is a Partition Group that allows this type tree to be used with varying types of transactions. For an example of the levels of UDF abstraction, see Figure 4-5.

]

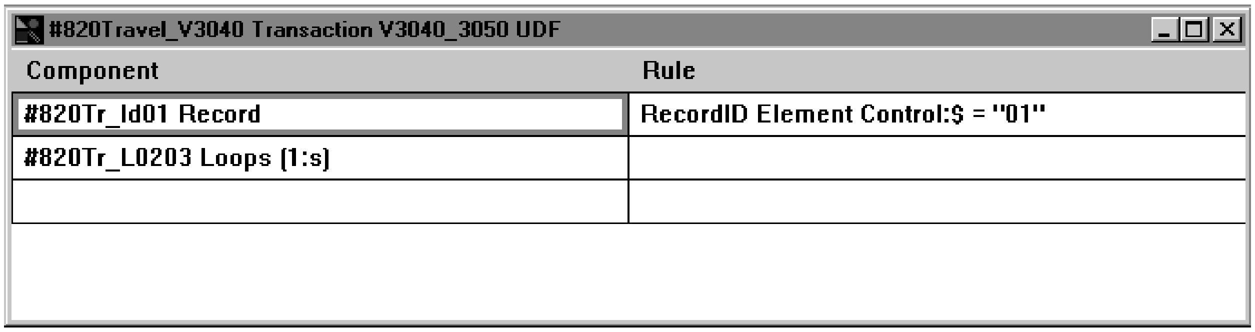
Figure 4-5 Levels of UDF Abstraction



Again, any category, group, or item may be added by copying from an existing type tree or by using the new element tool (leaf tool).

A Transaction is the series of elements that compose the UDF, usually in the form of a series of Records and/or Loops, as shown in Figure 4-6 (Rec1 and Rec2 Groups in Record Category). Loops are a series of Records that occur together and are repeated within the scope of the Transaction. Figure 4-6 illustrates the use of Records and Loops.

Figure 4-6 Record Loops



Component	Rule
#820Tr_Id01 Record	RecordID Element Control:$ = "01"
#820Tr_L0203 Loops (1:s)	

Components may be added to groups by double-clicking the target group and then dragging the elements to include as components into the desired position on the component list.

Figure 4-7 is a sample type tree in which the Loops partition groups have been expanded to reveal the type tree representations of the Loops used in this sample UDF. The Loops that comprise the #820Travel Transaction may be seen in their type tree representation in Figure 4-7.

Figure 4-7 Record Loops in Type Tree

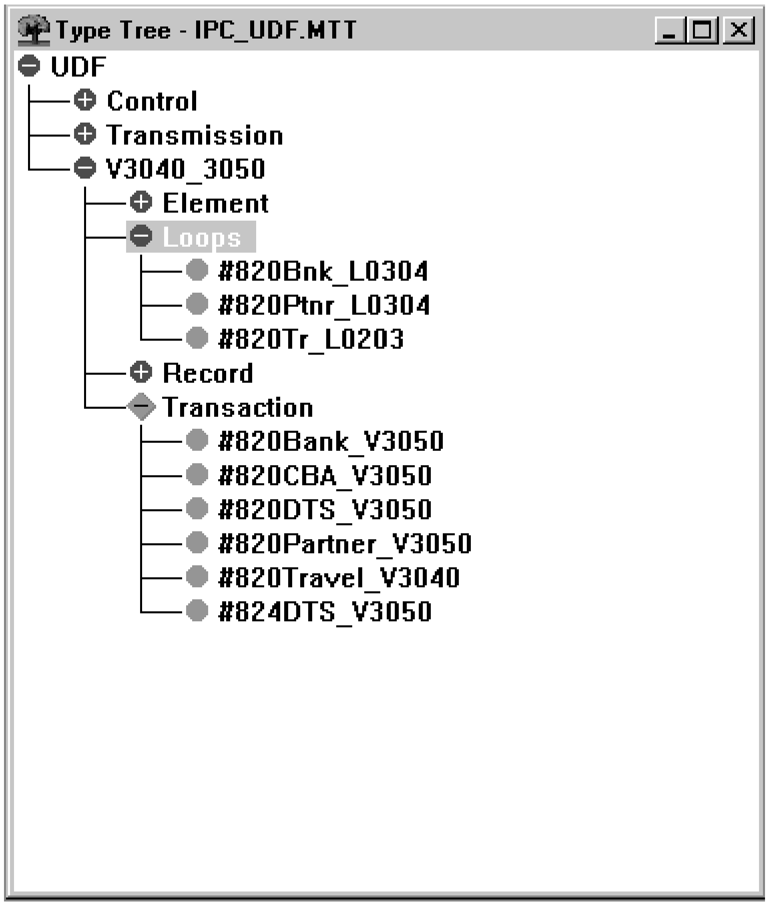
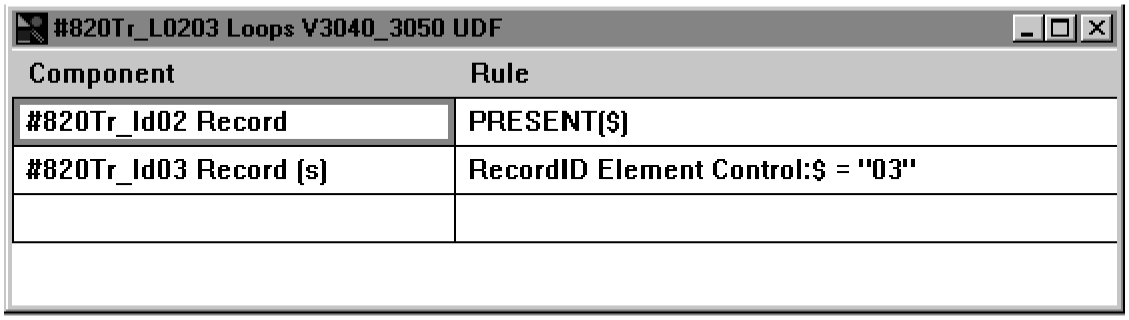


Figure 4-8, a Loop Component List, details a Loop used in this sample UDF demonstrating the concept of grouping multiple records and possibly multiple instantiations of records within a loop. Using Loops is optional, but Loops may be necessary to prevent long series of repeating records that appear in certain UDFs.

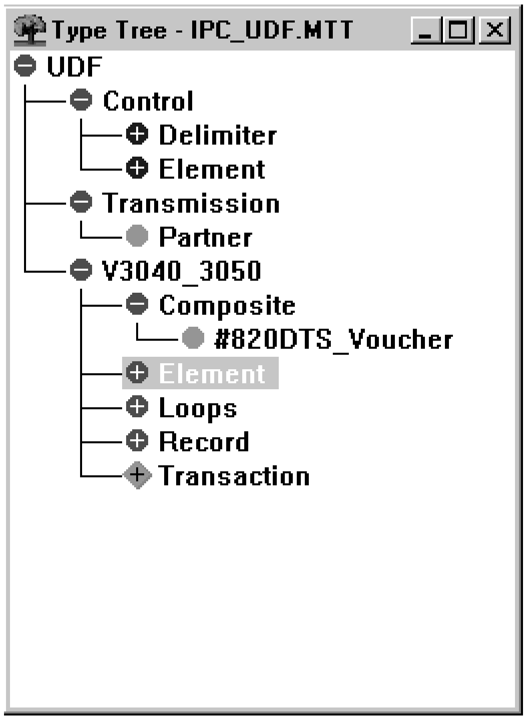
Figure 4-8 Loop Component List


#820Tr_L0203 Loops V3040_3050 UDF

Component	Rule
#820Tr_Id02 Record	PRESENT($)
#820Tr_Id03 Record (s)	RecordID Element Control:$ = "03"

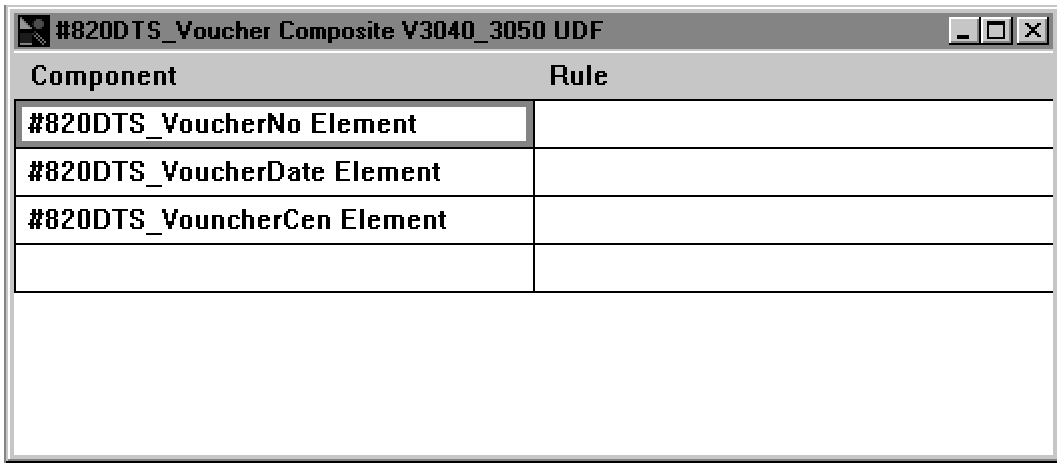
The components of the Records are defined in the UDF logical map specifications and appear as Elements or Composites of Elements in the type tree. Sometimes it is necessary to include more than one Element in a Composite. The Composite category is expanded in Figure 4-9, but the Element category is not, due to the length involved in so doing. Each Element would be abstracted as an Item whose attributes are defined by the UDF logical map specifications.

Figure 4-9 Composite List Type Tree



The abstraction, Composite, describes the basic grouping together of a series of related elements. The use of Composites is optional, but Composites may be used to denote a group of related elements, or of repeating groups of Elements whose existence is associated to other Elements within the group. In the example UDF, the Composite #820DTS_Voucher, shown in Figure 4-9, is used to group a series of elements related to each other and the specific Voucher, which the data describes. Figure 4-10 is the Component List of the #820DTS_Voucher Composite showing the Elements used to comprise the Composite.

Figure 4-10 Composite Component List



Component	Rule
#820DTS_VoucherNo Element	
#820DTS_VoucherDate Element	
#820DTS_VouncherCen Element	

4.4 Creating Other Type Trees

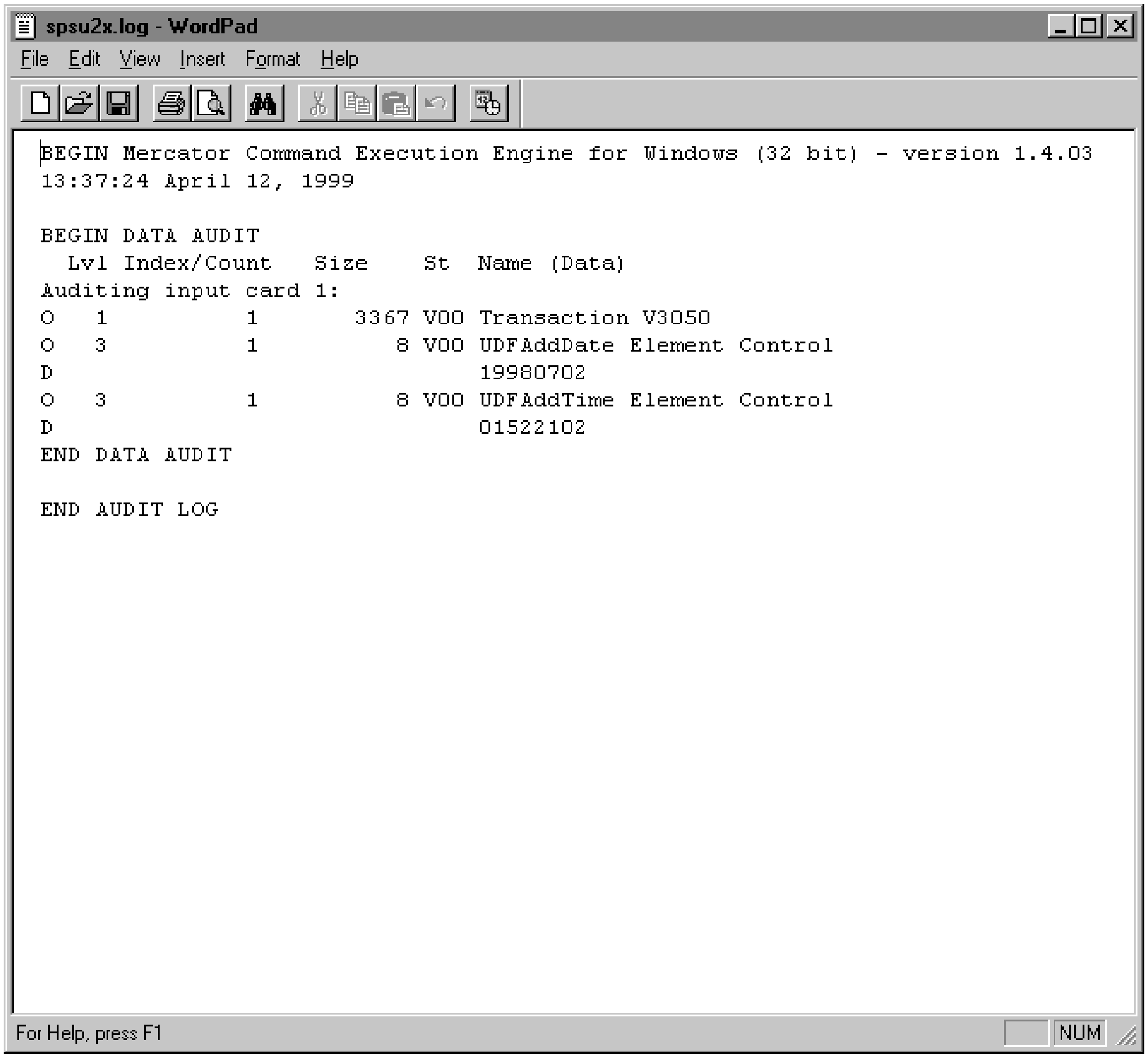
At times, it is necessary to create type trees that are specific to data structures other than UDF or X12 specifications. Other type trees that you may need to create are as follows: the premap type trees, the input type tree for the x824gen map, the input type tree for the x997gen map, and the type trees used to represent the data in an external lookup table. The need for these type trees depends upon your specific system requirements.

4.4.1 Acknowledgments

When creating x824gen and x997gen maps, you must have type trees that represent the data provided in the Mercator audit log. You should use an existing audit log type tree and modify it to fit the specific format of the data contained within the audit log. This audit log data is generated from the UDF-to-X12 map for x824gen or the X12-to-UDF map for x997gen. Figure 4-11 shows an example audit log.

When creating, changing, and using the type trees for translation error detection and reporting, you must have a basic understanding of the layout of the Mercator-generated audit logs. When referencing the audit log for the purpose of error reporting and translation acknowledgment, much of the data included in the audit log is irrelevant to the process and may be referred to as either placeholders or data used for distinguishability purposes.

Figure 4-11 Example Audit Log



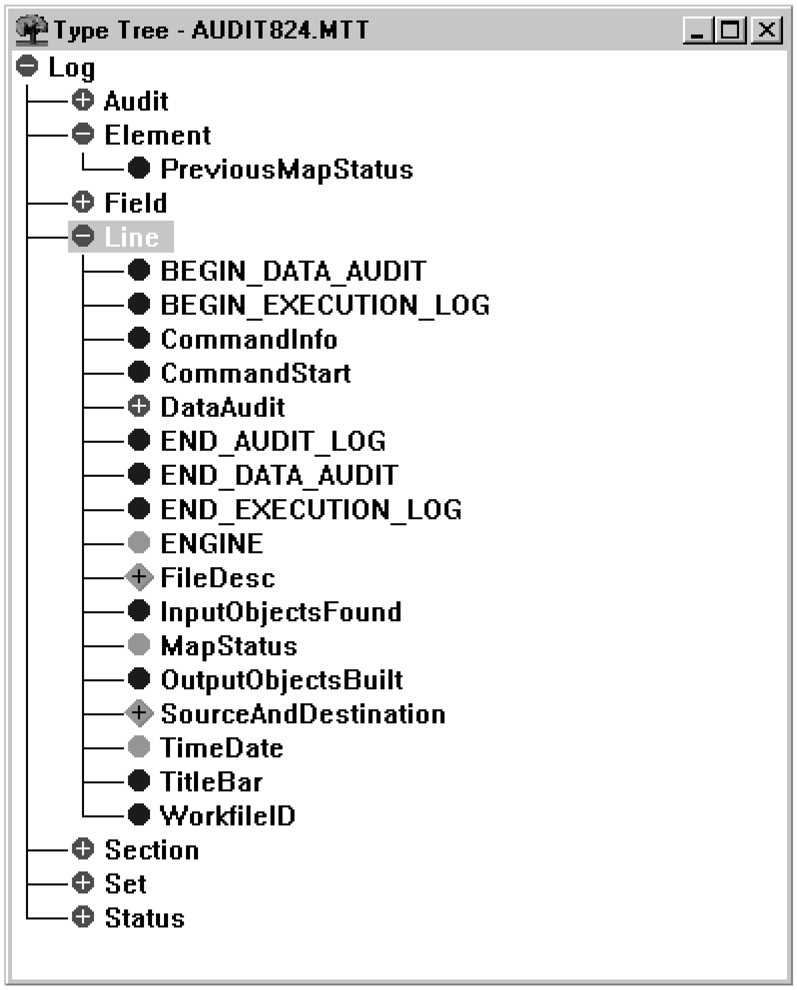
```
BEGIN Mercator Command Execution Engine for Windows (32 bit) - version 1.4.03
13:37:24 April 12, 1999

BEGIN DATA AUDIT
  Lvl Index/Count   Size    St  Name (Data)
Auditing input card 1:
O   1          1       3367 V00 Transaction V3050
O   3          1          8 V00 UDFAddDate Element Control
D                               19980702
O   3          1          8 V00 UDFAddTime Element Control
D                               01522102
END DATA AUDIT

END AUDIT LOG
```

At this time, the only data that is pertinent to the error reporting and translation acknowledgment process is the data contained within the data audit section, delineated by the BEGIN DATA AUDIT and END DATA AUDIT data markers (Figure 4-11). Because of the complexity of the type tree needed, you *should* use a pre-existing audit log type tree and edit that tree to meet your needs. The type tree displayed in Figure 4-12 shows the basic categories used, along with a breakdown of all of the data lines (Line category) necessary for containing the placeholders and data used for distinguishability purposes.

Figure 4-12 Audit Type Tree

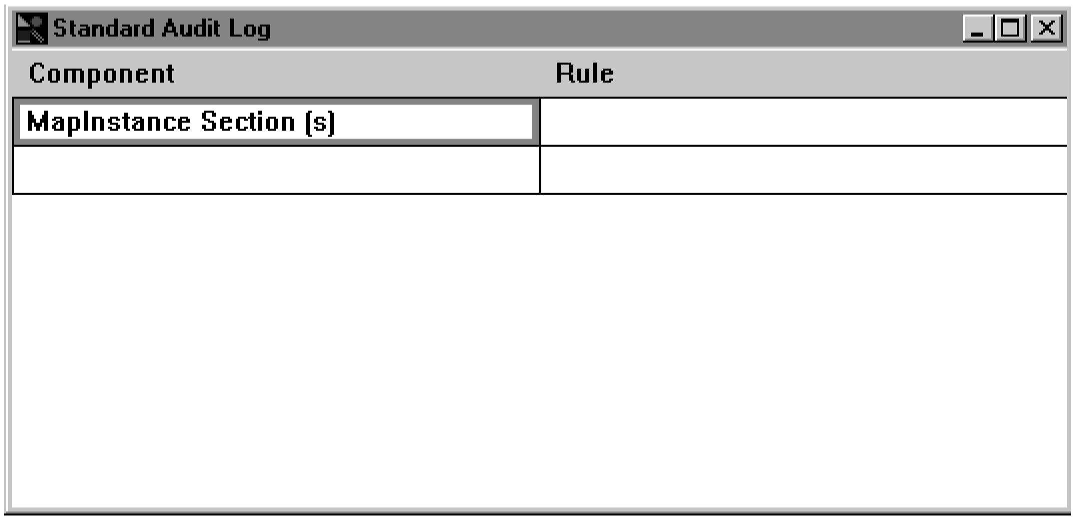


Type Tree - AUDIT824.MTT

```
Log
├─Audit
├─Element
│ └─PreviousMapStatus
├─Field
├─Line
│ ├─BEGIN_DATA_AUDIT
│ ├─BEGIN_EXECUTION_LOG
│ ├─CommandInfo
│ ├─CommandStart
│ ├─DataAudit
│ ├─END_AUDIT_LOG
│ ├─END_DATA_AUDIT
│ ├─END_EXECUTION_LOG
│ ├─ENGINE
│ ├─FileDesc
│ ├─InputObjectsFound
│ ├─MapStatus
│ ├─OutputObjectsBuilt
│ ├─SourceAndDestination
│ ├─TimeDate
│ ├─TitleBar
│ └─WorkfileID
├─Section
├─Set
└─Status
```

The Audit Log group is used as the top level of the data abstraction of the Mercator audit log. Figure 4-13 shows the components of the audit log. The audit log is abstracted as a series of 0-to-many instances of MapInstance Section.

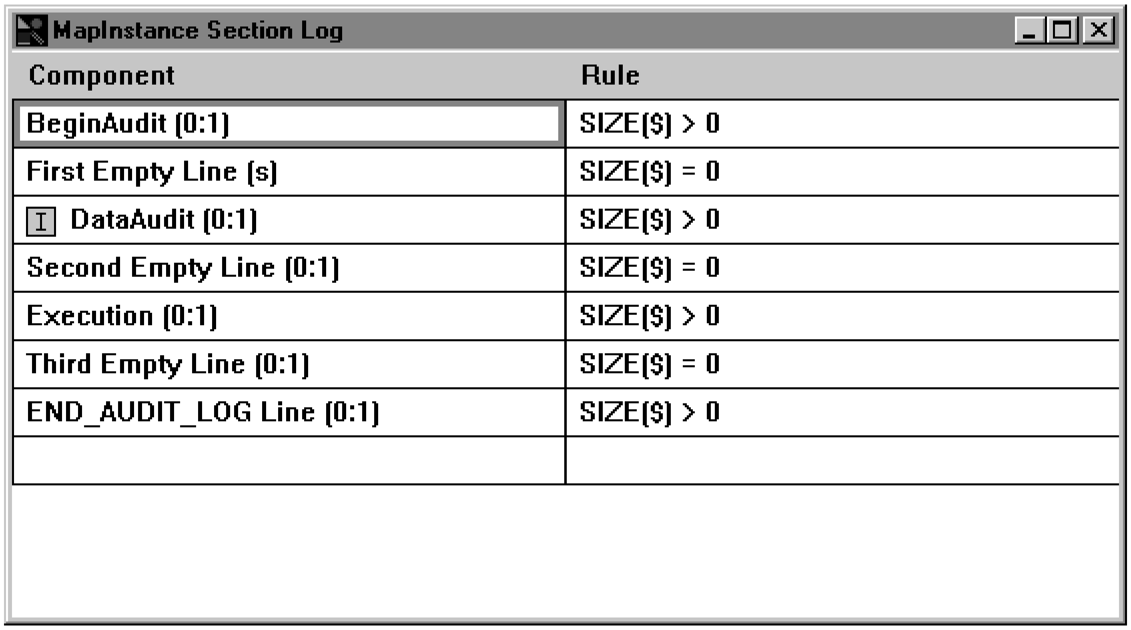
Figure 4-13 Audit Log Component List



Component	Rule
MapInstance Section (s)	

Figure 4-14 shows the items that compose the MapInstance Section container. The MapInstance Section container is indicative of the representation of the Mercator audit log layout. As previously mentioned, the majority of the components listed in Figure 4-14 are used to contain information pertinent to the execution of the original UDF-to-X12 mapping process.

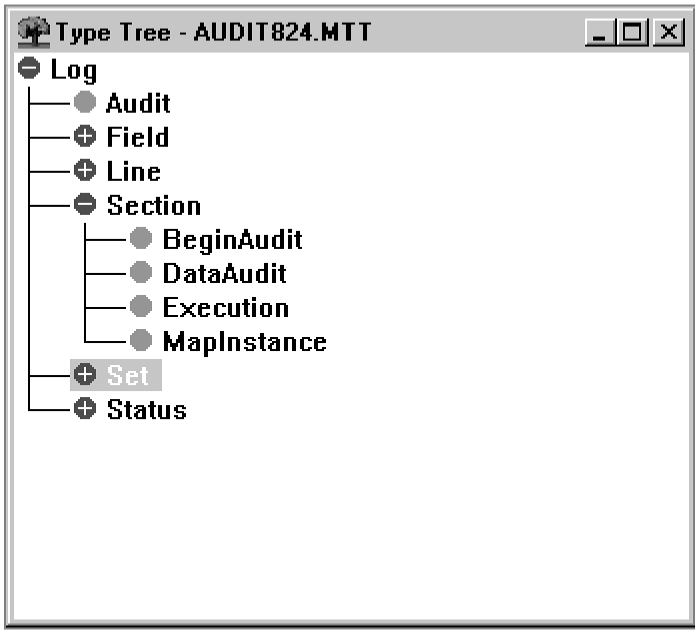
Figure 4-14 Map Instance Component List



Component	Rule
BeginAudit (0:1)	SIZE($) > 0
First Empty Line (s)	SIZE($) = 0
DataAudit (0:1)	SIZE($) > 0
Second Empty Line (0:1)	SIZE($) = 0
Execution (0:1)	SIZE($) > 0
Third Empty Line (0:1)	SIZE($) = 0
END_AUDIT_LOG Line (0:1)	SIZE($) > 0

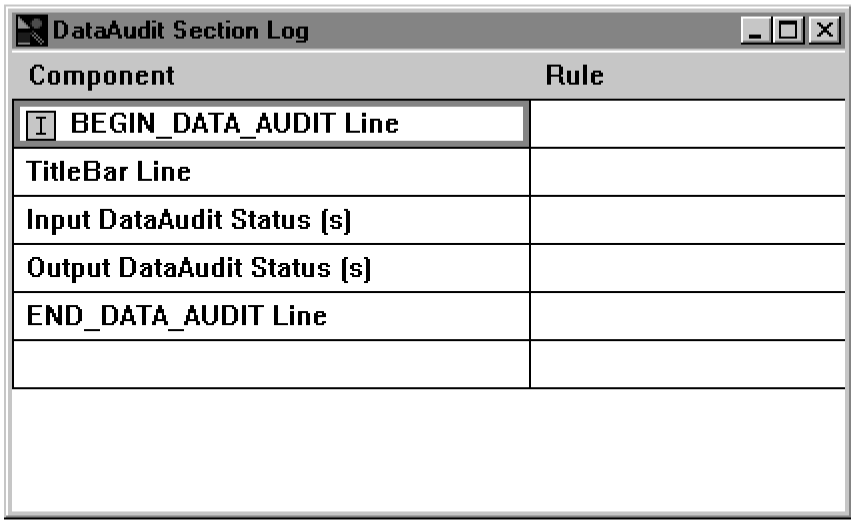
Figure 4-15 shows the groups contained within the Section category. The important groups are the MapInstance container (previously discussed in this section) and the DataAudit container. In order to view the contents of these or any groups in Mercator, double-click the group to view, and the Component lists are displayed.

Figure 4-15 Audit Type Tree Section List



Double-clicking the DataAudit Section group displays the component list, as shown in Figure 4-16. This list shows the basic makeup of the Data Audit portion of the Mercator audit log. Referring to the audit log displayed in Figure 4-11, note that the BEGIN_DATA_AUDIT Line component of this list refers to the line in the audit log that states BEGIN DATA AUDIT. The next line of the audit log beginning with “Lvl Index/Count” comprises the TitleBar Line. The line that follows “Auditing Input Card 1:” is the first element for the Input DataAudit Status portion of the DataAudit Section. The second element of that group is the Transaction Set.

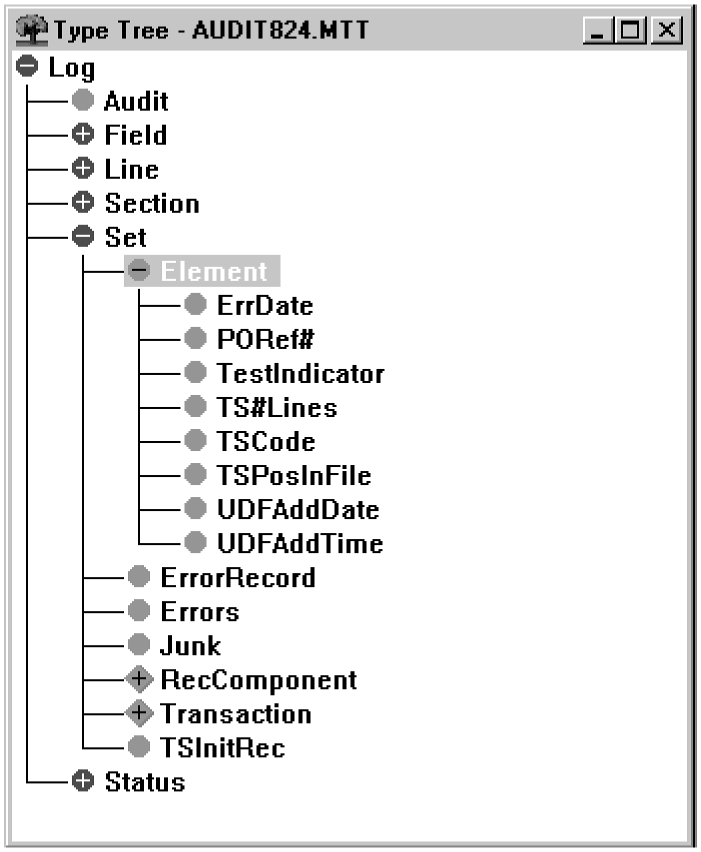
Figure 4-16 Data Audit Component List


DataAudit Section Log

Component	Rule
I BEGIN_DATA_AUDIT Line	
TitleBar Line	
Input DataAudit Status (s)	
Output DataAudit Status (s)	
END_DATA_AUDIT Line	

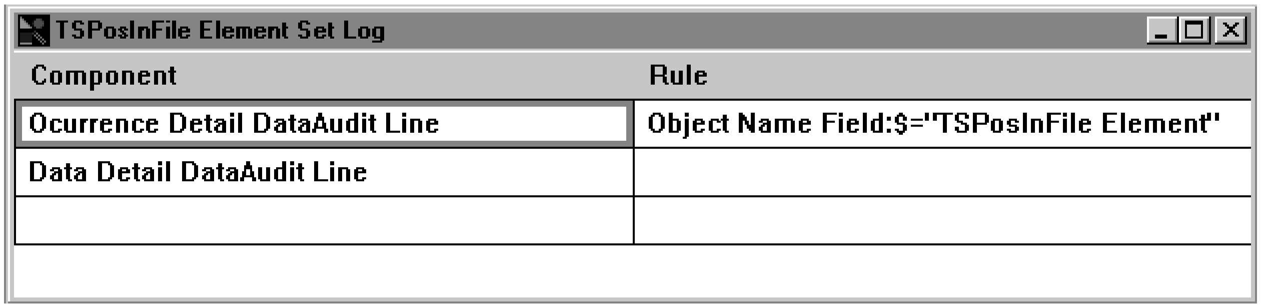
To this point, no major, if any, changes have been necessary to make this audit log type tree match the requirements that you have for a specific data audit. Figure 4-17 and Figure 4-18 display a list of elements that you need to change according to your data auditing requirements. The type tree shown in Figure 4-17 has the Element group expanded to show the list of elements that occur within the data audit section, in alphabetical order not order of occurrence.

Figure 4-17 Audit Type Tree Element List



Each Element is composed of the components displayed in Figure 4-18. You need to change the component rule for the Occurrence Detail DataAudit Line to reflect the data as it appears in the audit log.

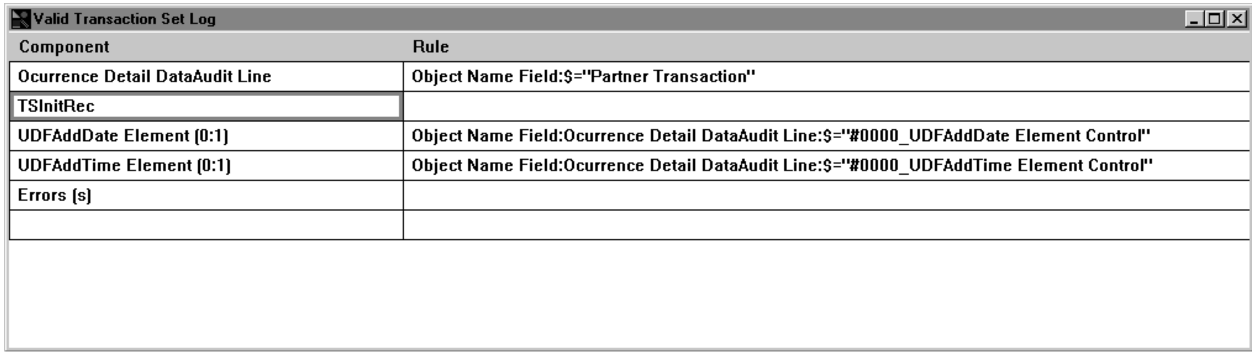
Figure 4-18 Audit Element Component List


TSPosInFile Element Set Log

Component	Rule
Ocurrence Detail DataAudit Line	Object Name Field:$="TSPosInFile Element"
Data Detail DataAudit Line	

To reflect the makeup of your audit log Data Audit Section, the Transaction Set components must be modified. Figure 4-19 shows the sample breakdown of the Transaction Set portion of this type tree. In your type tree, you should make changes to assure that you are properly identifying the Transaction level of the data audit, in this case titled "Partner Transaction". The Error component remains for your type tree. This section is actually used to report all of the errors that occur during the translation process. The Elements following the Error portion contain all of the data that is passed through the audit log from the original UDF. That data may or may not be recorded, as is evident by the (0:1) qualifier at the end of each Element, stating that there is at most one occurrence, but that the data may not exist. As discussed previously, these elements are added as you have data recorded in the audit log, and the component rules for each element are indicative of the data identified by each element.

Figure 4-19 Transaction Component List

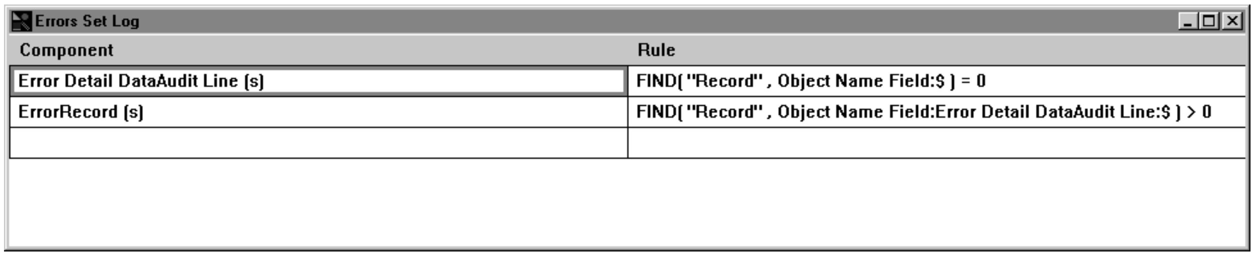


Valid Transaction Set Log

Component	Rule
Ocurrence Detail DataAudit Line	Object Name Field:$="Partner Transaction"
TSInitRec	
UDFAddDate Element (0:1)	Object Name Field:Ocurrence Detail DataAudit Line:$="#0000_UDFAddDate Element Control"
UDFAddTime Element (0:1)	Object Name Field:Ocurrence Detail DataAudit Line:$="#0000_UDFAddTime Element Control"
Errors (s)	

An Error Detail Line followed by a line that identifies the error and any possible erroneous data represents each error. The components of each Error Set are displayed below. The component rule for the Error Detail DataAudit Line states that there should be no occurrence of the word *Record* in the audit line, because Mercator should always report that an error has occurred within the Transaction level. The Error Record component states that there may be zero-to-many errors occurring at the Record level or below. The component rule states that the Record must appear at the next level for further error mapping to occur. Mercator will report that there was an error at the Record level before proceeding to report Element-level errors.

Figure 4-20 Error Component List

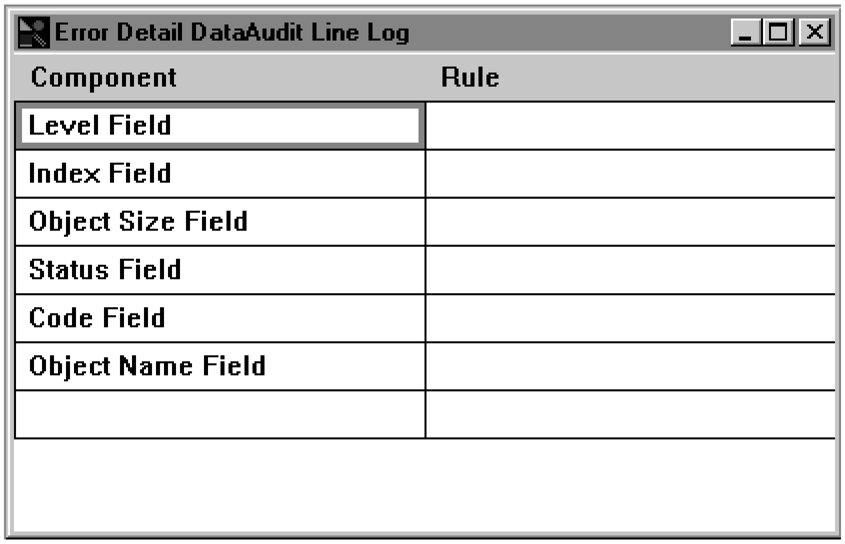


Errors Set Log

Component	Rule
Error Detail DataAudit Line (s)	FIND("Record" , Object Name Field:$) = 0
ErrorRecord (s)	FIND("Record" , Object Name Field:Error Detail DataAudit Line:$) > 0

As shown in Figure 4-21, each Error Detail DataAudit Line comprises different identification components. The Level Component is used to determine which level of processing an item (e.g., transaction, record, element) received, or if an error has occurred. The level is determined with respect to the input/output card. The card object is defined as Level 0. The Index Component is used to record either the index number of the object in a series of objects or a count of an object that can occur multiple times. The Size Component is used to determine the size in bytes of the object. The Status Component is used to determine the status of the object’s data with values (V - Valid, E - Error, W - Warning), and the Code Component is used to record a two-digit numeric code to further identify the status. The Object Name Component contains the name of the specific object. This component identifies each data line in the audit log and can be seen throughout the component rules above listed as Object Name Field.

Figure 4-21   Error Detail Component List

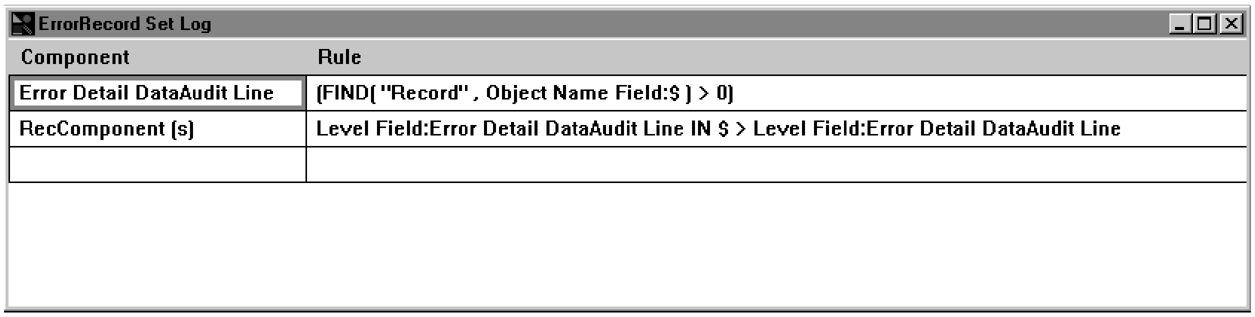


Error Detail DataAudit Line Log

Component	Rule
Level Field	
Index Field	
Object Size Field	
Status Field	
Code Field	
Object Name Field	

The Record level error reporting capabilities are broken down into the abstraction of ErrorRecord Set. Figure 4-22 displays a component list of the contents of the Error Record or Record level error reporting. Again, the data audit line begins with the Error Detail DataAudit Line, specifying the basic error information. The Error Record may or may not then be identified in the Record Component or RecComponent. This denotes any errors in the element or composite level of input/output data. The errors at the element or composite level have a higher Level number than those in the record, thus providing distinguishability between levels of error reporting.

Figure 4-22   Error Record Component List

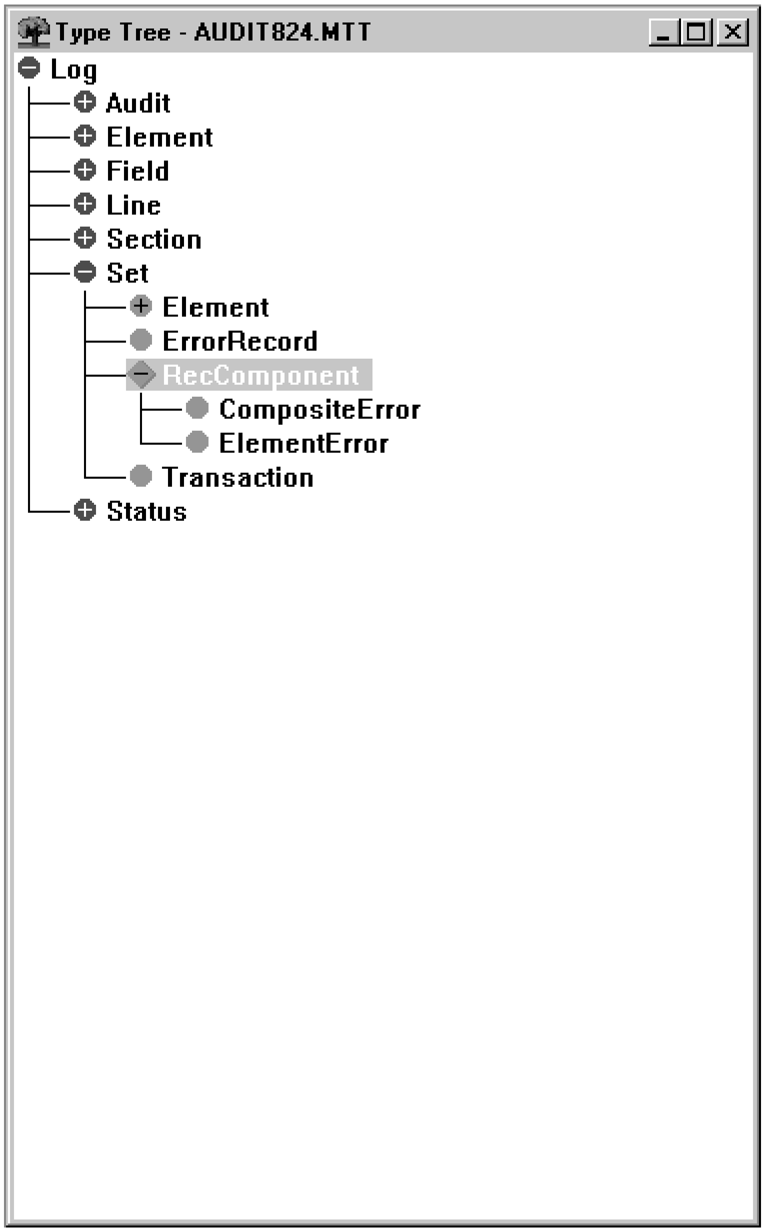


ErrorRecord Set Log

Component	Rule
Error Detail DataAudit Line	(FIND("Record" , Object Name Field:$) > 0)
RecComponent (s)	Level Field:Error Detail DataAudit Line IN $ > Level Field:Error Detail DataAudit Line

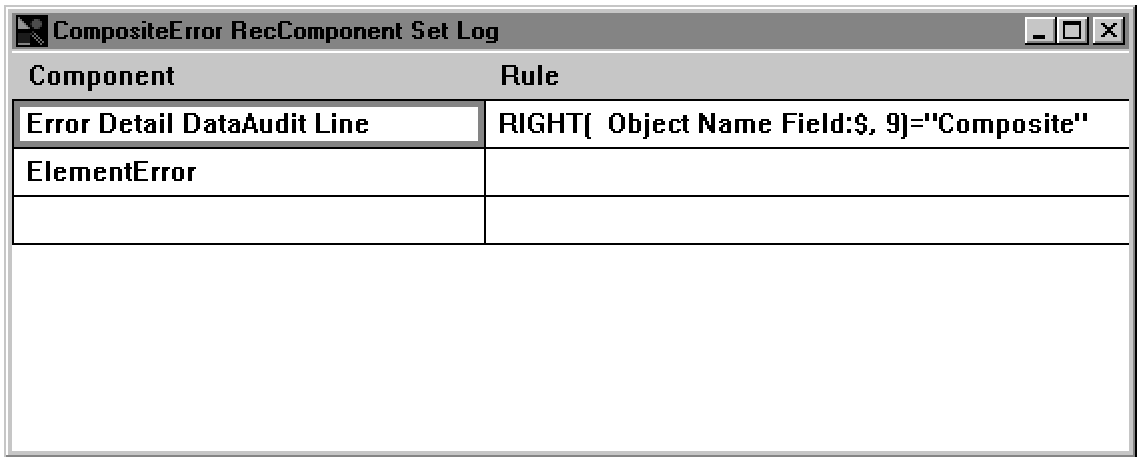
Figure 4-23 demonstrates the composition of the RecComponent as it relates to the Composite level and Element level errors.

Figure 4-23   Audit Type Tree Set List



The components of the CompositeError group are shown in Figure 4-24 to emphasize the changes necessary for identification of composite and element levels. The Object Name Field contains the word Composite or the equivalent word used in your file to identify a grouping of similar elements. In the case of Element level error reporting, the same line would contain the word Element or the equivalent word used in your file to identify your individual items or elements. Following the component identifying the error level, a data audit line appears specifying the data that is in error (if available) denoted by the ElementError component.

Figure 4-24   Composite Error Component List



CompositeError RecComponent Set Log

Component	Rule
Error Detail DataAudit Line	RIGHT(Object Name Field:$, 9)="Composite"
ElementError	

4.4.2 Premap Type Trees

The premap type tree(s) are used in the premap mapping process to break down a series of incoming transactions contained within one transmission into individual transactions per transmission. These type trees should be as simple as possible and still be able to adequately determine the beginning and end of each transaction within the transmission. At times it is necessary, depending upon the mapping and data validation requirements, to use a more detailed type tree. This detailed type tree is particularly necessary for those premap mapping processes that require padding or labeling of the UDF data.

The premap stage breaks the incoming UDF into individual transaction sets so that they can be passed into the UDF-to-X12 map one at a time. Some map families also require that the premap format the data. IPC uses the premap to make sure that the transaction sets that depend on each other are all present. SPS/ITIMP/APADE use the premap to pad records that are variable length. The premap should do very little, if any, data validation. Because the detailed 824 reports are generated from the UDF-to-X12 map's audit log, the premap should not fail. The type tree for the input card of the premap should have just enough information to determine where each transaction starts and ends.

The premap needs one or two output cards. If the data does not need any additional formatting, the premap only needs one output card to output the sizes for parsing the original UDF message. If additional formatting is needed, then the premap must have a second output card to output the modified data. If two output cards are needed, then the message description needs too include the premap_udf:yes token. The mapper should create the type trees for the input card and the modified data card. These type trees may be kept in the same type tree file since they use similar data elements.

The type tree that outputs the sizes may be reused from any of the other map families because it is a standard format across the map families. This sizes type tree contains a record for each transaction. The records are separated by linefeeds. Each record has three fields that are delimited by commas. The first field is required: It contains the size of the original UDF transaction. The second field may be used to indicate the type of transaction. This field is blank if the type of transaction does not affect the translation process. (IPC uses this field but SPS/ITIMP/APADE and SAACONS do not.) The third field indicates the size of the modified UDF. This field is blank if the data is not modified.

4.4.3 Additional Input Look-up Table Type Trees

Sometimes it is necessary to create a type tree that represents data contained within external look-up table files that are used during the mapping process. These type trees should be made from scratch, and care should be taken to assure that the type trees accurately reflect the data to be contained within this structure. The type tree should allow comment lines within the file, because the file is presented to a DEBX administrator for editing through the communications manager interface, and the rules for data entry should be specified in the top of the file as comments.

5.0 Map Construction

5.1 Translating UDF to X12

A logical map document specifies how elements from a UDF are mapped to an X12 and vice versa. This document is developed in coordination with the UDF system proponents. For some systems, this document is part of the UDF specification for that system. For other systems, such as in the DTS to DFAS scenario (since they neither receive nor send X12s) it is separately developed. The following sections detail the construction of a UDF-to-X12 map.

5.1.1 Map Naming Conventions

The convention for naming maps in the UDF-to-X12 map is to name all of the maps according to the X12 type tree. The executable map has a prefix of "E" to indicate that it is the executable map. The functional maps have a prefix of "f_" to indicate that they are functional maps. The names of the maps higher in the Navigator map tree view should reflect the X12 enveloping structure. Appropriate names for the ISA/IEA and GS/GE wrappers are "f_MapIC" and "f_MapFG". The functional maps for the transaction sets should have the transaction number in them. For example, the functional map for the 865-transaction set should be "f_865Sets". The functional maps for the segments should contain the transaction set number and the segment initiator. If the transaction uses a segment initiator more than once, the position number of the segment can be included. The position number can be found in the IC. Some examples are f_865_PKG and f_865_PWK210. The first example is a functional map for mapping to the Package segment, and the second example is a functional map for mapping to the paperwork segment with the position of 210. An "L" in the map name designates a loop. The 865-transaction set has a POC loop for its detail loop. The functional map name should be "f_LPOC". All segments in the loop should have the loop name included in their map name. The POC loop has a CTP segment. The functional map name would be "f_865_LPOC_CTP". Any segments that loop within the POC loop would also have another "L" to designate the loop. The 865 POC has a PID loop in it, the functional map name would be "f_865_LPOC_LPID". If the mapper needs to create a functional map to help convert values from the UDF format to the X12 the functional map name should be descriptive. For example the SPS map family's UDF-to-X12 map uses the "f_DecIt" and "F_NmbIt" functions to convert numbers that are being represented as text from the UDF representation to the X12 representation.

5.1.2 ISA Segment

The ISA Segment should be filled out as follows:

Table 5-1 ISA Segment

Element Delimiter	=SYMBOL(29)
Auth’nInfoQual’r Element	= “00”
Auth’nInfo Element	= NONE
SecurityInfoQual’r Element	= “00”
SecurityInfo Element	= NONE
Sender InterchangeIDQual’r Element	Use ISASNDRQUAL exitproc, e.g., =LEFT(EXIT(“libec_mapprocs.sl”, “map_isasndrqual”, “NULL”), 2)
InterchangeSenderID Element	Use ISASNDR exitproc, e.g., =LEFT(EXIT(“libec_mapprocs.sl”, “map_isasndr”, “NULL”), 15)
Receiver InterchangeIDQual’r Element	Use ISARECVQUAL exitproc.
InterchangeRcv’rID Element	Use ISARECV exitproc.
InterchangeDate Element	=CURRENTDATE()
InterchangeTime Element	=CURRENTTIME()
InterchangeCtrlStandardsID Element	=“U”
InterchangeCtrlVersion# Element	Fill in Control Version Number
InterchangeCtrl#	Use ICN exitproc and convert it to a number, e.g., =TEXTTONUMBER(TRIMRIGHT(LEFT(EXIT libec_mapprocs.sl”, “map_icn”, “NULL”),9)))
Ack’tRequested Element	=“0” (Unless otherwise specified by UDF)
TestIndicator Element	=“P” (Unless otherwise specified by UDF)
Composite Delimiter	=SYMBOL(31)
Terminator Delimiter	=SYMBOL(28)

The values mapped to the InterchangeRcv’rIDQual’r and the InterchangeRcv’rID are determined by the UDF.

5.1.3 GS Segment

The GS Segment should be filled out as follows:

Table 5-2 GS Segment

SegID Element	="GS"
Funct'lIDCd Element	Fill in according to IC of transaction set being mapped.
App'nSenderCd Element	Value from UDF if present.
App'nRcv'rCd Element	Value from UDF if present.
DateElement	=CURRENTDATE()
TimeElement	=LEFT(TIMETOTEXT(CURRENTTIME()), 2) + MID(TIMETOTEXT (CURRENTTIME()), 4,2) +RIGHT (TIMETOTEXT(CURRENTTIME()),2)
GroupCtrl# Element	Use GCN exitproc, e.g., =TEXTTONUMBER(TRIMRIGHT(EXIT("libec_mapprocs.sl" ,"map_gcn", "NULL"), 9)))
RspAgencyCd Element	="X"
VersionReleaseIndustryID Cd Element	Fill in Version number.

For more information on how to map the App'nSenderCd and App'nRcv'rCd Elements, refer to Section 3.4.2.

5.1.4 Audit Settings

The audit log should be as small as possible. There is no need to audit on valid segments because the 824 does not need to refer to segment indexes. The only information needed by the 824 map is any errors that were in the UDF. So the UDF-to-X12 maps need to report on the occurrence of any transactions and provide detailed information on errors in any transaction, and the item data in error. Assuming the UDF is mapping to the X12 3050 standard and that the UDF type tree follows the conventions laid out in Section 4.3, the audit settings should be set to audit "ANY Transaction V3050: UDF" with the "Track" set to "occurrence", the "Detail" set to "error", and the "Item Data" set to "error". The mapper can set additional audit settings if the data will be needed in the generation of the 824. Remember that if the incoming UDF has serious mistakes in it, the audit log might not be able to generate more information other than "Transaction V3050 UDF" in error.

5.2 Translating X12 to UDF

The following sections detail the construction of an X12-to-UDF map.

5.2.1 Map Naming Conventions

In the mapping process, the naming convention for all maps should be determined according to the destination element. In this case (X12 to UDF), the UDF type tree destination elements will be used. The executable map should begin with the prefix "E" to indicate that it is an executable map, while the functional maps should begin with the prefix "f_". The names of the maps dealing with the enveloping structure of the message should reflect the enveloping structure used in the UDF.

From a conceptual point of view, a typical transmission is a file composed of one or more transactions. There may or may not be multiple transaction types for any given UDF. A typical transaction is composed of one or more records or loops of records, which, in turn, is composed of one or more elements.

In the case of multiple transaction types being used in a UDF, a functional map may be necessary to handle the container of the multiple transaction types. For example, if the transaction types are grouped based upon X12 versions (e.g., V3050, V3070), a functional map may be required to map each version. The recommended convention for this example would be something like "f_MapV3070". It would follow that there would be a functional map necessary for each of the transaction types using a naming convention such as "f_Map810".

Within each transaction, a functional map would be necessary for each of the records or loops of records. These maps would adopt the naming described by the destination record or loop, such as "f_810_R001" or "f_810_L004006", where "R" is used to denote a record and "L" is used to denote a loop. In this example, "f_810_R001" would be used to map information to a 001 record, and "f_810_L004006" would be used to map information to a loop containing records 004 through 006. The transaction type (843 in this case) is also included to track the transaction in which the loop or record is used.

Within each record, a functional map may be needed to map individual elements or composites of elements. Again, the naming should reflect the destination element or composite, such as "f_810_R001_CPersInfo" or "f_810_R001_MailAddrs", where "C" is used to denote mapping to a composite. In this example, "f_810_R001_CPersInfo" would be used to map information to a composite named PersInfo, and "f_810_R001_MailAddrs" would be used to map information to an element named MailAddrs. Note the transaction type (810) and record (001) are maintained as part of the functional map name for ease of identification.

If, at the element level, further use of functional maps for converting, concatenating, calculating or other processes of manipulating the data is necessary, the naming convention should be used to best describe the process. For example, if a function map is needed to append a value for State to the end of a value for City, a name such as "f_AppendState2City" could be used.

5.3. Building the Premap(s)

An incoming UDF file may require one or two stages of preprocessing prior to passing it on to the main UDF-to-X12 map.

5.3.1 The Premap1 Stage

The (optional) premap1 map should take, as input, the original UDF file and rearrange the data within it so that there are single addressee contiguous messages within the output file. This is necessary for message types that use a list of address lines above a single body of message text (e.g., PADDS). The premap1 stage duplicates the text body after each address line, so that the resulting output file can be divided into single addressee messages.

5.3.2 The Premap Stage

The normal premap should take, as input, the original UDF file or the output from the premap1 stage if it is used. If the "premap_udf=yes" flag is set in the message description file, the premap is expected to create the following two output files: an index file that details the message bounding in the input file and UDF output file and the premapped UDF output file. If the "premap_udf=yes" flag is not set in the message description file, the premap is expected to create one output file, an index file that details the message bounding in the input file.

5.3.2.1 The Premap Index Output File

The index file is a list of three field records. Each record references a single message (transaction) in the input UDF file. The first record field is a number which refers to the size of the single transaction within the input UDF file. The second record field is a string that specifies the transaction type. (Note that this value is currently useful for IPC UDF-to-X12 mapping only). The third record field is a number which refers to the size of the single transaction within the output UDF file. The DEBX translator uses these values to divide the input and output UDFs into single transaction pieces.

5.3.2.2 The Premap UDF Output File

The UDF output file has all alterations necessary to make it possible to properly analyze the message in the regular UDF-to-X12 map. Some alterations that have been used include:

- Padding the lines out to a known number of spaces so that optional fields can be defined as fixed length in the UDF-to-X12 type tree.
- Globally filtering out, or replacing specific byte values as dictated by the message type rules.
- Placing record initiators on records to make them easily identifiable in the UDF-to-X12 map.

6.0 Addressing Procedures

Throughout this section, the Mercator "EXIT" function is referenced. For a complete discussion of this function, see Appendix A.

For *all* X12 messages produced by DEBX, the ISA sender ID qualifier/code (ISA05/06) corresponds to the DEBX system routing the message:

<ISA05> **= LEFT(EXIT("libec_mapprocs.sl","map_isasndrqual","NULL"), 2)**

<ISA06> **= LEFT(EXIT("libec_mapprocs.sl","map_isasndr","NULL"), 15)**

The LEFT function is necessary because of Mercator's inability to deal with the null terminators returned in an EXIT function data structure. The last input argument is ignored by the translator in both these cases, but Mercator requires all arguments to have some value to invoke the call. The string "NULL" has no particular significance, other than for readability and consistency between map families. The remaining sections only include ISA07/08 and GS02/03 to avoid redundancy.

6.1 Implicit Addressing

This section pertains to message types in which both sender and receiver ID codes can be found within the message body and cross-referenced using DEBX's trading partner database (TPDB), without referring to a separate table or external information such as a file name. The boldface type indicates portions of rules that should be entered verbatim, whereas italicized print indicates general instructions needing some interpretation depending on the specific map family. Also, map rules given for GS02/03 assume a maximum element length of 15 bytes, although some older X12 versions specify shorter lengths (e.g., Rev3010 allows only 12 bytes for each), and the corresponding rules need to be modified in those instances.

6.1.1 UDF-to-X12 Translation

The type of the receiver's ID code given by the UDF needs to be known for a TPDB lookup to get the values of ISA07/08. If a map family does not exclusively use a given ID type, the ID code can be distinguished by its characteristics. The qualifier values correspond to those used in element #66 Identification Code Qualifier, which appears in the N1 segment (among other places). To avoid confusion, note that these values are not identical for all qualifier elements in the X12 standard; for example, element #128 Reference Identification Qualifier denotes a CAGE code using "W7", instead of the qualifier "33" used for addressing purposes.

Table 6-1 Trading Partner Identifier Types and Properties

Type name	Length	Properties	Qualifier
DUNS	9	Numeric	1
DUNS+4	13	Numeric	9
DODAAC	6	Alphanumeric	10

Type name	**Length**	**Properties**	**Qualifier**
CAGE	5	1st and 5th Positions Numeric, Middle 3 Positions Alphanumeric Excluding the Letters "I" and "O"	33

For map families in which incoming UDFs are translated to outgoing X12s, such as SPS and SAACONS, the interchange receiver values should correspond to the receiver specified in the UDF. The map rules for ISA07/08 are:

<ISA07> **= IF(** *<UDFrecvID>* **= "PUBLIC" , "ZZ" ,**

LEFT(EXIT("libec_mapprocs.sl","map_isarecvqual" , " *<UDFrecvIDqual>* | *<UDFrecvID>"* **) , 2))**

<ISA08> **= IF(** *<UDFrecvID>* **= "PUBLIC" , "PUBLIC" ,**

LEFT(EXIT("libec_mapprocs.sl",map_isarecv" , " *<UDFrecvIDqual>* **|** *<UDFrecvID>"* **) , 15))**

where *<UDFrecvIDqual>* is the appropriate qualifier based on the above table. The conditional statements bypass the TPDB lookups if the message receiver is "PUBLIC".

In contrast, map families which always produce an outgoing UDF (i.e., UDF to X12 to UDF, or "U2X2U"), including ADS and DIFMS, should have ISA07/08 identical to ISA05/06 since the system is technically sending the X12 message to itself for retranslation into a UDF:

<ISA07> **= LEFT(EXIT("libec_mapprocs.sl","map_isasndrqual","NULL"), 2)**

<ISA08> **= LEFT(EXIT("libec_mapprocs.sl","map_isasndr","NULL") , 15)**

In both cases, the GS02/03 values should be mapped directly from the UDF without conversion:

<GS02> = *<UDFsndrID>*

<GS03> = *<UDFrecvID>*

6.1.2 X12-to-UDF Translation

The sender and receiver in the UDF should be mapped respectively from the GS02 and GS03 of the incoming X12, without conversion:

<UDFsndrID> = *<GS02>*

<UDFrecvID> = *<GS03>*

6.1.3 X12 997 Acknowledgment from X12-to-UDF Translation Audit Log

The audit log for the X12-to-UDF map should be set up to include all sender/receiver information from the original X12 message, except for U2X2U families which do not require X12-to-UDF error reporting. Because the 997 acknowledgment is returned to the sender of the original message, its addressing information can be mapped from the audit log of the original message in reverse order (excluding ISA05/06 as previously described):

```
<ISA07> = <ISA05 of incoming X12, from audit log>
<ISA08> = <ISA06     "        "       "     >
```

and

```
<GS02> = <GS03 of incoming X12, from audit log>
<GS03> = <GS02     "        "       "     >
```

6.1.4 X12 824 Acknowledgment from UDF-to-X12 Translation Audit Log

Unlike an incoming X12 which will have its addressing information verified by the system before translation, it cannot be assumed that a UDF will have adequate information to route an acknowledgment. Therefore, the sender is specified as the DEBX system for the GS02 (in addition to ISA05/06), and the receiver (in both GS03 and ISA07/08) is denoted using the name of the channel from which the original UDF was received. The latter is provided using the EXIT function INCHANNEL and qualified with “ZZ”:

<ISA07> **= "ZZ"**

<ISA08> **= LEFT(EXIT("libec_mapprocs.sl","map_inchannel", "NULL"), 15)**

and

<GS02> **= TRIMRIGHT(LEFT(EXIT("libec_mapprocs.sl","map_isasndr","NULL"), 15)))**

<GS03> **= TRIMRIGHT(LEFT(EXIT("libec_mapprocs.sl","map_inchannel", "NULL"), 15)))**

6.2 External Addressing – SAACONS Example

Certain UDFs require information external to the message content to identify the sender and/or receiver. For example, a SAACONS message only contains the commercial trading partner’s ID, identifying the Government site via a three-digit file name extension. A separate table is cross-referenced to find the site ID (DODAAC in this case) based on the extension. The implications of using an external look-up approach are outlined in this section, beginning with some general guidelines:

- The type tree representing the lookup table should allow for comments/faulty entries/blank lines without ignoring valid data or failing translation. One approach is to create a partitioned "Record" group.
- Whenever referring to table entries in the Mercator LOOKUP function, use the TRIMLEFT function to remove any leading spaces. (Mercator considers leading spaces in a left-justified field to be part of the data, which would affect comparisons).

6.2.1 UDF-to-X12 Translation

A functional map is used to perform the external lookup for GS02, as a workaround to the limitation that EXIT functions cannot be nested:

```
=f_LookupDODAAC( RIGHT(
LEFT(EXIT("libec_mapprocs.sl","map_remotefilename","NULL"),
                        30), 3), SAACONS_DB)
```

where the right-most 3 bytes of the UDF file name (i.e., the extension) and the entire SAACONS look-up file are the input arguments. The rule used in the functional map is:

```
=EITHER( TRIMLEFT( LOOKUP( SiteIDCdElement:.:SAACONS_DB ,
            TRIMLEFT(SiteFilenameIDCdElement:.:SAACONS_DB) = FN_ext_ID ) ) ,
       TRIMRIGHT( LEFT(EXIT("libec_mapprocs.sl","map_lookupfail",
          ( "UDF Filename Extension Site-ID  """ +
             FN_ext_ID + """  not found in " + GETFILENAME(SAACONS_DB) ) +
                          " |            " ) , 12 ) ) )
```

where SiteIDCdElement and SiteFilenameIDCdElement denote the DODAACs and corresponding file name extensions in the table, respectively, and FN_ext_ID is the actual extension. If no match is found (i.e., the LOOKUP evaluates to NONE) the EITHER function causes the FAIL function to be invoked, which triggers the same error condition resulting from a TPDB look-up failure. The first argument is an error message displayed to the DEBX administrator, which includes the unrecognized file name extension and the name of the lookup table for troubleshooting purposes. The last argument, consisting of 12 blanks, is the value returned by "FAIL".

6.2.2 X12 824 Acknowledgment from UDF-to-X12 Translation Audit Log

The 824 acknowledgment would not require any external addressing if it were being sent as an X12 message, as is the case with the 997 acknowledgment. However, because it will be translated back into UDF and returned to the original sender, the filename extension from the original incoming UDF must be preserved. The extension is used instead of the corresponding DODAAC because an 824 should be generated and returned to the original sender regardless of whether it can be found in the current look-up table:

```
="[" + RIGHT( LEFT(EXIT("libec_mapprocs.sl","map_remotefilename","NULL"), 30 ), 3 ) + "]"
```

Enclosing the value in brackets will enable the X12-to-UDF map to distinguish it as a file name extension rather than a DODAAC, which would normally be expected in GS03.

6.2.3 X12-to-UDF Translation

To produce the correct file name extension, the function SETSITEID is called to set the value of the DEBX file name variable “sid”. The corresponding map rule is used where GS03 is written to Record00:

```
=IF( LEFT( App'nRcv'rCd Element:GS , 1 ) = "[" ,
   LEFT(EXIT("libec_mapprocs.sl","map_setsiteid",MID( App'nRcv'rCd Element:GS , 2 , 3 )  +
      “ | “ + App'nRcv'rCd Element:GS + "              "  ) , 15 ) ,
   LEFT(EXIT("libec_mapprocs.sl","map_setsiteid " , EITHER( TRIMLEFT(
       LOOKUP( SiteFilenameIDCd Element:.:SAACONS_DB ,
            TRIMLEFT(SiteIDCd Element:.:SAACONS_DB) = GS03 ) ) ,
                         SYMBOL(0) )  + “ | “ +
           GS03 + "              " ) , 15 ) )
```

The conditional checks to see whether GS03 is enclosed in brackets, indicating the message is an 824 acknowledgment, in which case it passes the extension value directly. Otherwise, the value is assumed to be the DODAAC of the receiving SAACONS site and is cross-referenced to obtain the correct extension. If the LOOKUP fails, the null character is passed which will automatically trigger an error condition with an appropriate error message. Because SETSITEID returns its last input argument, the variable-length GS03 is padded to ensure that LEFT truncates the null-terminator.

6.3 Combined Implicit/External Addressing – IPC Example

Although the IPC UDF contains all sender and receiver information within the message body, the trading partner (receiver) could be identified using either a CAGE code or a DSSN. Because the latter is not included in the TPDB, an external look-up table is required.

6.3.1 UDF-to-X12 Translation

The addressing for ISA07/08 is basically a two-step process. The receiver’s ID is first assumed to be a CAGE code, so the map rule for ISA07 is:

```
=IF( PRESENT( LEFT(EXIT("libec_mapprocs.sl","map_isarecvqual" , "33 | “ +
              MID( #820Ptnr_VendorID Element:.:UDF , 2 , 5 ) ) , 15 ) ) ,
    LEFT(EXIT("libec_mapprocs.sl","map_isarecvqual " , "33 | “ +
              MID( #820Ptnr_VendorID Element:.:UDF , 2 , 5 ) ) , 15 ) ,
    f_VTISA07( UDF , VendorInfo ) )
```

The conditional verifies a successful TPDB lookup, in which case the returned value is mapped to ISA07. If the lookup fails, the code is assumed to be a DSSN, and the map f_VTISA07 is called to perform the external lookup:

```
=EITHER( f_Reset( LOOKUP( ISA07 Element Vendor:.:VendorInfo,
 #820Ptnr_DSSN Element:.:UDF = DSSN Element Vendor:.:VendorInfo  &
 #820Ptnr_VendorID Element:.:UDF = Vendor_ID Element Vendor:.:VendorInfo ) ) ,
        f_Reset("{none}") )
```

If the LOOKUP successfully matches the ID to a DSSN match in the table, the corresponding value for ISA07 is passed to f_Reset; otherwise, the map is invoked with “{none}” to indicate a look-up failure. Finally, f_Reset determines whether to reset the error condition triggered by the original TPDB look-up failure (using RESETERROR) depending on the result of the external lookup:

```
=IF( In1 = "{none}" , NONE , LEFT(EXIT("libec_mapprocs.sl","map_reseterror","NULL"), In1 + "
" ) , 2 ) )
```

Similar steps are taken to obtain ISA08.

7.0 Integration Procedures

This appendix describes how to add an externally developed map family to the DEBX software. For each new map family (UDF message type) you create, you may add a package of maps and supporting files to a DEBX system that has been previously loaded with a translation segment provided by INRI. This package you add should consist of the following components:

- A set of maps for the UDF message type (described in Section 3.0) to perform:
  - UDF-to-X12 translation
  - X12-to-UDF translation
  - (optional) UDF-to-X12 premap1 to preprocess UDF input
  - (optional) UDF-to-X12 premap to preprocess UDF input
  - (optional) 824 X12 (UDF-to-X12 acknowledgment) generation
  - (optional) 997 X12 (X12-to-UDF acknowledgment) generation
- Any additional look-up tables needed for input to the above UDF-to-X12 translation and X12-to-UDF translation.
- (optional) Map family help documents, with a main index HTML document referencing any other provided documents.
- A message description file detailing the names and processing rules for the map family (described in Section C), and the optional HTML index file for online help.

While it is possible to install a new map family manually, it is best to create a package with an accompanying installation script to do the installation. This allows the DEBX Administrators to reinstall the new map family after a new translation segment is loaded and eliminates errors.

The following installation script should be used to install your package. Copy it into a file, make it executable (e.g., chmod a+x install_maps.sh), add your file names within the quotes of the variables, place it into a directory with your files, and archive the files into a package (a tape archive).

```
#!/bin/sh

# This script installs a map family on a DEBX system that has
# been loaded with a translation segment.
# It should reside in the same directory with the map,
# message description, documentation, etc. files that
# are getting installed.
# PLACE YOUR MAP FAMILY NAME WITHIN QUOTES. THIS MUST ALSO BE
# THE NAME OF YOUR MESSAGE DESCRIPTION FILE:
# e.g.
# MAP_FAM="MY_UDF"

MAP_FAM=""

# PLACE YOUR MAPS (*.hp files) WITHIN QUOTES:
# e.g.
# MAP_FILES="x2u.hp u2x.hp 824gen.hp 997gen.hp premap.hp"
```

```
MAP_FILES=""

# PLACE YOUR LOOKUP TABLES (*.tbl files) WITHIN QUOTES:
# e.g.
# LOOKUP_TABLES="accnt2addr.tbl accnt2name.tbl"

LOOKUP_TABLES=""

# PLACE YOUR DOC (*.html and *.pdf files) WITHIN QUOTES:
# e.g.
# DOC_FILES="my_udf_index.html my_udf_spec.pdf my_udf_ref.pdf"

DOC_FILES=""

# ----- END USER MOD SECTION ------------

InstallFiles() {
      if [ "X$filelist" != "X" ]; then
            for f in $filelist
                  do
                  if [ ! -e ./$f ]; then
                        echo "$f does not exist in local directory."
                        echo "Exiting with ERROR."
                        exit 1
                  fi

                  if [ ! -d $dest ]; then
                        mkdir -p $dest
                  fi
                  echo "copying $f to $dest"
                  cp  ./$f $dest
                  chown $owner $dest/$f
                  chmod $mode $dest/$f
            done
      fi
}

# Check for root user to run...
user_name=`whoami'
if [ x$user_name != "xroot" ] ; then
      echo "You must be root to run this script."
      exit 1
fi

mode=664
owner=DEBX:hawk

filelist=$MAP_FAM
dest=/h/data/global/EC/Messages/MessageDesc
```

```
InstallFiles

filelist=$MAP_FILES
dest=/h/data/global/EC/Messages/Maps/$MAP_FAM

InstallFiles

filelist=$LOOKUP_TABLES
dest=/h/data/global/EC/Messages/Maps/$MAP_FAM

InstallFiles

filelist=$DOC_FILES
dest=/h/data/local/EC/html/MapDocs/$MAP_FAM

InstallFiles

# Put the map family name in TOC file if not already in it:
in_there=`grep -c "^${MAP_FAM}$"
/h/data/global/EC/Messages/MessageDesc/TOC’

if [ "X$in_there" = "X0" ]; then
      echo $MAP_FAM >> /h/data/global/EC/Messages/MessageDesc/TOC
fi
```

Once the package is created, it may be installed on a loaded DEBX system by extracting the archive and running the install script (as root). You must stop and restart the DEBX software before the new map family is available for channel configuration.

Appendix A EXIT Functions

Mercator provides the ability to temporarily “exit” the map to perform external processing, and then reenter the map to continue processing. This functionality is invoked using the Mercator EXIT function when the EXIT function is invoked. The map passes processing back to the DEBX translator (temporarily), and the DEBX translator processes the request and passes the results back to the map. This Appendix includes all EXIT functions that are supported in DEBX Version 3.0. The general usage of an EXIT function is =EXIT(Function_Name, Arg1, Arg2). Mercator requires values for both arguments to invoke the function. Unused arguments are denoted by the string “NULL”.

Note that several exitprocs use a parameter qualifier when accessing the trading partner database. The list of these values is as follows:

01 or 09	DUNS
33	CAGE
M3	DSSN
10	DODAAC

map_838found

Indicates whether an 838 *Trading Partner Profile* was found in the TPDB for the most recent 843 *Response to RFQ* received; the match is based on the value of GS02. This is a SAACONS-specific function.

USAGE: EXIT("libec_mapprocs.sl","map_838found", "NULL")

RETURN VALUE: "Y" or "N", depending on whether an 838 was found.

map_append_xlog

This function appends the passed string to the daily (channel-based) translation log. The string is time and date stamped by the translator. The daily (channel-based) translation logs are used for map family specific message reporting capabilities, in which the message reporter invokes a handler to parse these daily logs to create daily, weekly, and/or monthly data reports, as needed.

USAGE: EXIT("libec_mapprocs.sl","map_append_xlog",arg-string)

RETURN VALUE: "NULL"
The arg-string is the string to append to the translation log.

map_auditlog

Instructs the translator to append the passed string to the audit log at translation completion. These strings are accumulated during the translation. At the end of translation, the audit log is appended with the following format.

```
START_ADDL_AUDIT
<string1>
<string2>
```

```
.
.
.
<stringn>
END_ADDL_AUDIT
```

This functionality is useful in a sequence of translations where the audit log will be used in a subsequent map (e.g., x824gen map).

USAGE: EXIT("libec_mapprocs.sl","map_auditlog",arg-string)

RETURN VALUE: The string placed in arg-string. The value for arg-string is the string to append to the Audit Log.

map_filetor

Returns the time of receipt of the received file being translated.

USAGE: EXIT("libec_mapprocs.sl","map_filetor","NULL")

RETURN VALUE: Date of Receipt in YYYYMMDDHHMMSS format.

map_gcn

Obtains a site-specific unique (sequential) functional group control number.

USAGE: EXIT("libec_mapprocs.sl","map_gcn", "NULL")

RETURN VALUE: 9 bytes/left-justified/space-padded

map_get_var

This function retrieves a string variable's string value. It can be called by any map in a translation series to get a variable's value. For the variable value set functionality, see map_set_var.

A translation sequence is defined by the series of translations done to create an X12 or UDF message, and its associated acknowledgments or ancillary parts (e.g., the 838 message that accompanies an 843 during X12-to-UDF SAACONS translation). For UDF-to-X12 translations, the premap(s) are also part of the sequence. This function can set a variable in one part of a map and retrieve that value in another part of the same map. Also, this function can pass values on to the next map in a sequence (e.g., the premap can set a variable, and then the UDF-to-X12 map can access that value).

USAGE: EXIT("libec_mapprocs.sl","map_getvar",variable)

RETURN VALUE: value of given variable or "NULL" if not set.

map_gsaddr

Performs a TPDB lookup to obtain the functional group ID code of a trading partner, given some other ID code value and qualifier.

USAGE: EXIT("libec_mapprocs.sl","map_gsaddr","qualifier | code")

RETURN VALUE: 15 bytes/left-justified/space-padded
Where qualifier is the Trading Partner ID code qualifier using one of the following values: "01", "09", "33", "M3", or "10", and code is the Trading Partner ID code.

map_icn
Obtains a site-specific unique (sequential) interchange control number.

USAGE: EXIT("libec_mapprocs.sl","map_icn","NULL")

RETURN VALUE: 9 bytes/left-justified/space-padded

map_inchannel
Returns the name of the channel on which the message was originally received.

USAGE: EXIT("libec_mapprocs.sl","map_inchannel", "NULL")

RETURN VALUE: 19 bytes/left-justified/space-padded

map_isarecv
Performs a TPDB lookup to obtain the interchange ID code of a trading partner, given some other ID code value and qualifier.

USAGE: EXIT("libec_mapprocs.sl","map_isarecv","qualifier | code")

RETURN VALUE: 15 bytes/left-justified/space-padded
Where qualifier is the Trading Partner ID code qualifier using one of the following values: "01", "09", "33", "M3", or "10", and code is the Trading Partner ID code.

map_isarecvqual
Performs a TPDB lookup to obtain the interchange ID code qualifier of a trading partner, given some other ID code value and qualifier.

USAGE: EXIT("libec_mapprocs.sl","map_isarecvqual","qualifier | code")

RETURN VALUE: 2 bytes/left-justified/space-padded
Where qualifier is the Trading Partner ID code qualifier using one of the following values: "01", "09", "33", "M3", or "10", and code is the Trading Partner ID code.

map_isasndr
Returns the ID code of the DEBX system as defined in the System Setup database or the channel-specific value if present.

USAGE: EXIT("libec_mapprocs.sl","map_isasndr","NULL")

RETURN VALUE: 15 bytes/left-justified/space-padded

map_isasndrqual

Returns the ID code qualifier of the DEBX system as defined in the System Setup database or the channel-specific value if present.

USAGE: EXIT("libec_mapprocs.sl","map_isasndrqual","NULL")

RETURN VALUE: 2 bytes/left-justified/space-padded

map_launchmap

Invokes an alternative map and set of rules for a designated portion of a message. The current translation will be completed, and then the alternate translation will be invoked. Currently, the launchmap function works only as a replacement data mechanism.

USAGE: EXIT("libec_mapprocs.sl","map_launchmap", "<token>|LOC=<message portion>,ERR=<error string>")

RETURN VALUE: "NULL"

Where:

token: A symbolic name for the launchmap rule. The value must match a token in the message description lm_list (see Appendix B for a complete description of the lm_list syntax).

message portion: A string designating the piece of the message on which to invoke the launchmap rule. It can have one of several forms:
- "ALL"
- "GCN=ALL"
- "GCN=123;STN=ALL"
- "GCN=123;STN=456"

The first two forms designate the whole message (as passed to the original map) should be passed to the new map in accordance with the launchmap rule. The third form designates that the entire functional group that has the GCN value of 123 should be passed to the new map in accordance with the launchmap rule. The forth form designates that only the transaction designated by STN 456 within GCN 123 should be passed to the new map in accordance with the launchmap rule.

Note: The designated message portion will be wrapped in an ISA and, if necessary, a GS envelope prior to being passed to the alternative map.

error string: A character string that will be placed into the JDS in front of the message portion string within the errors list. If no error string is provided, then the default will be: "Alternative map invoked to process:"

Example usage:

```
EXIT ("libec_mapprocs.sl","map_launchmap", "997FF_LOOKUP|LOC=GCN:120;STN:240,
          ERR=lookup failure: CAGE 2576")
```

map_lookupfail

Forces a message failure condition for a database or table look-up failure.

USAGE: EXIT("libec_mapprocs.sl","map_lookupfail","error-string | echo-string")

RETURN VALUE: Value of echo-string. Value for error-string will be attached to the associated message object, echo-string is the value to return back from function.

map_msn

Returns the message sequence number (MSN) of the message being translated (outgoing X12-to-UDF translations only).

USAGE: EXIT("libec_mapprocs.sl","map_msn","NULL")

RETURN VALUE: SNNNNNNNN/YYYYMMDD
where S = the single-character site ID
NNNNNNNN = the numeric (0 padded) sequence #
YYYYMMDD = the date value.

map_outchannel

Returns the name of the outgoing channel. (X12-to-UDF translation only)

USAGE: EXIT("libec_mapprocs.sl","map_outchannel", "NULL")

RETURN VALUE: 19 bytes/left-justified/space-padded

map_prevxstatus

Returns the translation status code of the previous translation. This function is useful when the map is in a sequence of translations (e.g., the 824 Audit Log map after a UDF-to-X12 translation).

USAGE: EXIT("libec_mapprocs.sl","map_prevxstatus", "NULL")

RETURN VALUE: a string representation of the return code from the previous translation.

map_remotefilename

Returns the original name of the file that contained the message being translated. This file name is the name of the file as it was received on the DEBX system.

RETURN VALUE FORMAT: 30 byte/left-justified/space-padded

USAGE: EXIT("libec_mapprocs.sl","map_remotefilename","NULL")

RETURN VALUE: 30 bytes/left-justified/space-padded

map_reseterror

Resets the current error status of the map. The recommended method for resetting the status in Version 3.0 is to use the map_tpdblookupnoerror function.

USAGE: EXIT("libec_mapprocs.sl","map_reseterror","NULL")

RETURN VALUE: "NULL"

map_setiteid

Sets the value of the <sid> file name variable, which is exclusively used for file name site identification; currently, only SAACONS uses this function.

USAGE: EXIT("libec_mapprocs.sl","map_setsiteid","id | echo-string")

RETURN VALUE: Value of echo-string.
  id is the value to assign to the site-id file name variable.
  echo-string is the value for the function to return back.

map_set_var

This function sets a string variable's value. It can be called by any map in a translation series to set a variable name/value pair, so that later, the variable's value can be requested during that same translation sequence. For the variable value request functionality, see map_get_var. A translation sequence is defined by the series of translations performed to create an X12 or UDF message, and its associated acknowledgments or ancillary parts (e.g., the 838 message that accompanies an 843 during X12-to-UDF SAACONS translation).

For UDF-to-X12 translations, the premap(s) are also part of the sequence. This function can be used to set a variable in one part of a map and to retrieve that value in another part of the same map. Also, this function can be used to pass values to the next map in a sequence (e.g., the premap can set a variable, and then the UDF-to-X12 map can access that value, or the X12-to-UDF map can set a variable, and the X12 997gen map can access that value.)

USAGE: EXIT("libec_mapprocs.sl","map_setvar","var-token | var-value")

RETURN VALUE: "NULL"
  var-token is the name of the variable to set.
  var-value is the value to assign to the variable.

map_tpdblookup

Performs a TPDB lookup to obtain some type of ID code of a trading partner, given some other ID code value and qualifier. If the lookup fails, the translation status is set to LOOKUP_FAILURE. For a non-destructive TPDB look-up, see TPDBLOOKUPNOERR.

USAGE: EXIT("libec_mapprocs.sl","map_tpdblookup","qualifiers | id code")

RETURN VALUE: 15 bytes/left-justified/space-padded
qualifiers is the comma-seperated string of the qualifier pairing, given-qual and requested-qual. given-qual is the qualifier of the given ID code and requested-qual is the qualifier for the requested ID code. Valid values are of the following: "01", "09", "33", "M3", and "10". id code is the given Trading Partner ID code.

map_tpdbllookupnoerr

Performs a TPDB lookup to obtain some type of ID code of a trading partner, given some other ID code value and qualifier.

USAGE: EXIT("libec_mapprocs.sl","map_tpdblookupnoerr","qualifiers | id code")

RETURN VALUE: 15 bytes/left-justified/space-padded
qualifiers is the comma-seperated string of the qualifier pairing, given-qual and requested-qual. given-qual is the qualifier of the given ID code and requested-qual is the qualifier for the requested ID code. Valid values are of the following: "01", "09", "33", "M3", and "10". id code is the given Trading Partner ID code.

map_transfail

Forces a message failure condition for a translation failure.

USAGE: EXIT("libec_mapprocs.sl","map_transfail","error-string | echo-string")

RETURN VALUE: Value of echo-string.
Value for error-string will be attached to the associated message object, echo-string is the value to return back from function.

map_varname

Used to set a transmit file name variable from the map. The <varname> portion is actually the variable to set. Example: VAR=SITENAME would set the SITENAME variable for transmit file name resolution. For a description of transmit file name resolution, see the Help system.

USAGE: EXIT("libec_mapprocs.sl", "map_var", "varname | value")

RETURN VALUE: "NULL"
varname is the name of the variable to set and value is the value to assign it.

Appendix B Message Description Files

The message description file contains fields that are used by the DEBX software to control map execution and transmit file name resolution. It also contains a full description field that is displayed to the DEBX administrator during channel configuration.

Each message description field used by the DEBX software and its usage is as follows:

- addl_inputs

A comma-separated list of file names (and possible locations) that will be provided to the u2x_map and the x2u_map during invocation. Each file can have an additional attribute of ";LOCK" following it. If the lock token is set for a file, then the translator locks this file before passing it to the map. An input file that will be updated by the map should always use the lock token to prevent other parallel translation events from accessing or updating the file while it is unstable. (See note below)

An example addl_inputs line:

addl_inputs: lookup1.tab, lookup2.tab;LOCK, lookup3.tab

The second look-up table is locked by the translator prior to passing to the map. It is unlocked after the map completes. This is a cooperative lock, where the file is only protected from multiple access if all accessors use the lock token. So if an input file is to be used by more than one map family, then each map family that accesses this file should use the lock token for that file.

- addl_outputs

A comma-separated list of file names (and possible locations) that will be provided to the u2x_map and the x2u_map during invocation. It is only useful during X12-to-UDF translation, where each additional output will be sent to the same destination as the regular (generated) UDF. (See note below)

- docs

This is an html file that serves as the table of contents page in a web browser when the user chooses to view the UDF and X12 specifications for this system.

- fulldesc

A full description of the map family to be displayed to the DEBX administrator during channel configuration.

- lm_list

A list of launch map tokens, and their associated configuration rules. Each member of the lm_list should have the following elements defined:

token: A string that provides the cross reference between the launch map rule, and the exitproc call to map_launchmap (see Appendix A for a complete description on invoking the map_launchmap exit proc).

mapfile: map filename to invoke *

inputs: any combination and number of %INPUT, %AUDIT, <additional lookup table names> *

The value of %INPUT designates that the portion of message content that is defined in the map_launchmp exit proc call should be properly wrapped in an envelope, and passed to the new map.

A value of %AUDIT designates that the audit logfrom the previous (main) translation should be passed to the new map.

output: The only currently supported value is REP_DATA, which means "replacement data". This means that the result of the launchmap invocation will replace the message portion, and will be sent out the destination channel.

ack_stat: Indicates how the associated acknowledgment message should be treated - as an ACK or a NACK for admin routing.

err_level: A setting that dictates the error level to assign to the message object for the purpose of error queue population. A level of ERR goes into the error queue, and a level of WARN does not.

* Note: Map and additional input file names and locations can be specified by using a full path and name or a name only. Alternatively, any or all of the path and/or name can be specified using variables that are evaluated at execution time.

- premap

The file name (and possible location) of the premap map to be invoked prior to the u2x_map. (See note below)

- premap1

The file name (and possible location) of the premap1 map to be invoked prior to the regular premap. (See note below)

- premap_udf

A "yes" or "no" value. If set to "yes", an output UDF file is expected when running premap (in addition to the index file). The output UDF is used as input to the u2x_map.

- reject838x824_map

A SAACONS-specific map to be invoked on the 843 X12 file when the x2u838_map fails, or there is no associated 838. (See note below)

- reports

A list of the reports available for generation. Each entry includes information about the report name, a token identifier, the frequency of the report, and an optional mime-type value. The report is then available for configuration from the admin tab of the channel configuration window.

- report_generator

The filename and location of the executable to be invoked for report generation.

- split_route

This field has the following possible values: "ISA", "GS", or "ST". It defines the outgoing (X12-to-UDF) translation granularity. This granularity defines the amount of the message that gets passed to the X12-to-UDF maps for a single translation. Through the channel configuration, the DEBX administrator can refine the granularity to a lower level (where ISA is the highest, and ST is the lowest). But the DEBX administrator cannot raise the granularity above the value that is set in this field. Note that since the default value for a UDF message type is "GS", if the field is not present in the message description file, the X12-to-UDF map will receive a single GS (and possibly multiple STs within it) per invocation.

- u2x_map

The file name (and possible location) of the UDF-to-X12 map file to be invoked. (See note below)

- variables

This is a semicolon separated list of variable names that is presented to the DEBX administrator for transmit file name building. The format of each element is a readable string, followed by a curly-brace-enclosed variable name. The X12-to-UDF map is expected to set the value of each of these variables during translation by using the VAR=<var name> exitproc.

- x2u_map

The file name (and possible location) of the X12-to-UDF map file to be invoked. (See note below)

- x2u838_map

A SAACONS-specific map to be invoked on the 838 X12 file for transmit along with an 843 message. (See note below)

- x824_map

The file name (and possible location) of the X12 824gen map file to be invoked after the UDF-to-X12 translation. (See note below)

- x997_map

The file name (and possible location) of the X12 997gen map file to be invoked after the X12-to-UDF translation. (See note below)

NOTE: Map, additional input, and additional output file names and locations can be specified by using a full path and name or a name only. Alternatively, any or all of the path and/or name can be specified using variables that are evaluated at execution time. These are not the same variables as those specified in the "variables" message description, above. These variables, and their expansion rules are as follows:

"%datafiles" – Expands to the value of the environment variable: VIDS_DATAFILES or to /h/data/global/EC if VIDS_DATAFILES is not set.

"%channel" – Expands to the value of the channel name. For instance, during X12-to-UDF translation for the SAACONS FTP1 channel, if the SAACONS message description has the value cross_ref_%channel.tab: in the addl_inputs list, then the file /h/data/global/EC/Messages/Maps/SAACONS/cross_ref_FTP1.tab is passed to the X12-to-UDF map at invocation.

"%map_fam" – Expands to the value of the message type.

After path/name expansion, if there is no leading slash ('/'), then the file is assumed to be in the $VIDS_DATAFILES/Messages/Maps/<map family name> directory. For the SAACONS map family, on a production or test platform, this would be /h/data/global/EC/Messages/Maps/SAACONS

The message description file is object based. An object is something that is enclosed in {}'s and has elements. The elements themselves are other objects, but with names (named objects). For example, here is an object with three elements:

```
{
 one   { 1 }
 two   { 2 }
 three { 3 }
}
```

Objects can be elements of other objects. For instance, the named object "three" could be an aggregate object, consisting of "four" and "five".

```
{
 one   { 1 }
 two   { 2 }
 three {
   four { 4 }
   five { 5 }
 }
}
```

An object can be a list, in which case it has a list of unnamed objects of the same type. While the objects are unnamed, they may have text preceeding the "{" (such as indices), which is ignored.

```
{
 0: {
   foo { 1 }
   bar { 2 }
 }
 1: {
   foo { 10 }
   bar { 20 }
 }
 2: {
   foo { 100 }
   bar { 200 }
 }
}
```

Named objects can exist in any order inside of their parent object, but list items are stored in the order that they are listed.

Here is the GTN message description, with comments (lines beginning with "#"), but comments are not yet supported in the real file (except before the first "{").

```
# the beginning of the entire object, which is the message description# itself; unnamed
{
# normal variable/value pairs
        premap_udf       { no }
        x2u_map          { gtnx2u.hp }
        x997_map         { gtnx997.hp }
        report_generator { /h/EC/progs/rpt_gen_gtn }
        docs             { gtndocs.html }
        shortdesc        { GTN }

# a list with only one element
        addl_inputs {
                {
                        name { gtncarr.tbl }
                        flags { 0 }
                }
        }

# a list with multiple elements
        lm_list {
                {
                        token { 997FF_LOOKUP }
                        mapfile { gtn_997ff.hp }
# a list inside of a list
                        inputs {
                                { %INPUT }
                                { gtncarr.tbl }
                        }
                        output { REP_DATA }
                        ack_stat { ACK }
                        err_level { ERR }
                }
                {
                        token { 997FF_XLATE }
                        mapfile { gtn_997ff.hp }
                        inputs {
                                { %INPUT }
                                { gtncarr.tbl }
                        }
                        output { REP_DATA }
                        ack_stat { NACK }
                        err_level { WARN }
                }
        }

# another list
        reports {
                {
# string types may be enclosed in quotation marks, but it is not necessary
                        name { "Daily Data Quality (text)" }
                        identifier { DQ_TXT }
                        frequency { daily }
                }
                {
                        name { "Daily Data Quality (html)" }
                        identifier { DQ_HTML }
                        frequency { daily }
                        mime_type { html }
                }
                {
                        name { "Daily Data Timeliness" }
                        identifier { DT_TXT }
                        frequency { daily }
                }
        }

# a large string value
        fulldesc {
"NAME
        GTN -- Global Transportation Network

        TRANSACTION TYPES
        Transaction Type                    UDF->X12    X12->UDF
        ----------------------------------------------------------------
        214 Transp. Carrier Shipment Status Message             X
        315 Status Details (Ocean)                      X

<bunch of stuff deleted>

REFERENCES
                Type  IC specification      IC version #  UDF specification    UDF date
```

```
==========================================================================
            214   NONE               NONE         CEDI RevC.doc       03/08/99
            ---------------------------------------------------------------------
            315   ts315(3060).doc       13153060      CEDI RevC.doc       03/08/99
"
      }
# the end of the message description object
}
```

Appendix C TESTAPI Map Family

The Mercator EXIT functions, as defined in Appendix A, can help you understand the map to translator API. To assist with this, the TESTAPI map family is provided. This map family can be used, as a translation type, so a DEBX system can be configured to receive a UDF message on a TESTAPI type channel or to send an X12 message to a TESTAPI type channel. Although the message being received or transmitted is the primary input to each of the TESTAPI maps, the additional input and additional output files provide the real value as follows:

- The additional input file, exitproc.tbl, contains records, each of which include the name of the EXIT function to invoke, the parameters to pass to the EXIT function, and a flag indicating whether the EXIT function should be invoked for UDF-to-X12 translation, X12-to-UDF translation, or both. The initial version of this file contains all of the EXIT functions and the associated parameters. In addition, through the edit channel window/edit look-up tables function, the functions (as defined in Appendix A) can be removed or different parameters can be used to further exercise each EXIT function (e.g., both success and failure conditions).

- The additional output file, results.tbl, will be generated for each translation and will include each record line from the additional input file with the results (as passed back to the map) appended.

The following test datafiles are provided in the /h/data/global/EC/Messages/Maps/ TESTAPI directory:

- sample.x12 – data used for an X12-to-UDF message translation (map supports V4010 -824, 836, 840, 843, 850, 855, 860, 865).

- sample.udf – data used for a UDF-to-X12 message translation.

- tpdb_filler – data used for creating the trading partner database to perform lookups.

- exitproc.tbl – data used for an X12-to-UDF or UDF-to-X12 message translation and contains a list of all EXIT functions to be invoked, along with the parameters to pass to each.

Format and usage of exitproc.tbl:

Each line has four fields that are delimited by semicolons. The first three fields are required. The first field must contain a flag value of "U2X", "X2U", or "BOTH", indicating which direction(s) of translation to invoke the the exitproc. Any invalid value in the first field will disable the exitproc. The second field contains the name of the library used for the exitproc. The third field contains the name of the exitproc. The fourth field, which is optional, contains any arguments passed to the exitproc. If arguments are not given, by default, this field will be set to "NULL". No spaces are necessary; any spaces preceding or following any of the field values will be disregarded. Any line not matching this format will cause the map to fail.

On a system loaded with DEBX (at least) version 3.0 and DEBX Translation segment (at least) Version 1.1, the following configuration is required. For detailed instructions for each of these steps, see the Help system.

- Configure the system setup database as described in the Help system.

- Make TESTAPI accessible through the Communications Manager by adding the line
  "TESTAPI"
  in the file /h/data/global/EC/Messages/Maps/TOC. Be sure to stop and re-start the DEBX Software to put this change into effect.

- Create an FTP channel to pull in the CCR data. The node type MUST be CCR, which is an option you must chose at channel creation time. Make the "pull from path" point to a directory you create, and the file name pattern should be "tpdb_filler".

- Copy the /h/data/global/EC/Messages/Maps/TESTAPI/tpdb_filler file to your created "pull from" directory.

- Turn on your CCR channel. This will pull and remove the "tpdb_filler" file (the remove operation is the reason you did not "pull" it directly from the original location). The messages within the file will be parsed, and the trading partner database (TPDB) will be populated. This will take a few minutes. You may view the TPDB from the databases menu option. Select Refresh (CTRL-R), to watch it be populated. When it stops populating, the processing is complete.

- Create an X12 channel to pull in an X12 message. The node type should be anything except CCR, and the protocol should be FTP. The "pull from path" should be a temporary directory you create, and the file name pattern should be "sample.x12". Turn off the "check reply route" option. The "push to path and name" is required, but can be anything (e.g. "/tmp/{ccc}").

- Copy the /h/data/global/EC/Messages/Maps/TESTAPI/sample.x12 to your temporary directory.

- Create a TESTAPI type UDF channel to pull the sample UDF file in, and to send the sample X12 message out (after X12-to-UDF translation). The "pull from path" should be a temporary directory you create, and the file name pattern should be "sample.udf". The "push to path and name" should be to the same temporary directory, with the name portion (after the last slash character): "/tmp/my_dir/out_udf.{ccc}".

- Copy the /h/data/global/EC/Messages/Maps/TESTAPI/sample.udf file to this temporary directory.

- Add a route using the route database window so that all messages received on the X12 channel will be routed to the TESTAPI UDF channel.

  You are now ready to begin the test:

- Turn on the TESTAPI UDF channel to pull in the sample.udf file. This will invoke the incoming translator, and all the EXIT functions listed in the exitproc.tbl file. Copy the /h/data/global/EC/Messages/Maps/TESTAPI/results.tbl file to a safe location for analysis. (The next translation will overwrite it).

- Turn on the X12 channel to pull in the sample.x12 file. The router will route it to the TESTAPI UDF channel. This will invoke the outcoming translator, and all the EXIT functions listed in the exitproc.tbl file. Copy the /h/data/global/EC/Messages/Maps/ TESTAPI/results.tbl file to a safe location for analysis. (The next translation will overwrite it).

  Repeat the test runs as many times as you wish by editing the exitproc.tbl input datafile to produce different results.

Appendix D Notes

The following acronyms and abbreviations appear in this document:

ADS: Automated Disbursing System

AIS: Automated Information System

ANSI: American National Standards Institute

APADE: Automated Procurement and Accounting Data Entry

API: Application Programming Interface

CAGE: Commercial and Government Entity Code

CCR: Central Contractor Registry

DAPS: Defense Automated Printing System

DEBX: DOD E-Business Exchange System

DIFMS: Defense Industrial Financial Management System

DISA: Defense Information Systems Agency

DODAAC: Department of Defense Activity Address Code

DODAAN: Department of Defense Activity Address Number

DSSN: Disbursing Station Sequence Number

DTS: Defense Travel System

DUNS: Data Universal Numbering System

EC: Electronic Commerce Infrastructure

ECI: Electronic Commerce Interchange

ECPN: Electronic Commerce Processing Node

EDI: Electronic Data Interchange

FTP: File Transport Protocol

GCN: Group Control Number

GUI: Graphical User Interface

GW: Gateway

HTML: Hypertext Markup Language

ICN: Interchange Control Number

INRI: Inter-National Research Institute

IPC: Integrated Paying and Collecting

ITIMP: Integrated Technical Item Management and Procurement

MSN: Message Sequence Number

PADDS: Procurement Automated Data and Document System

RFQ: Request for Quotation

SAACONS: Standard Army Accounting and Contracting System

SABRS: Standard Accounting, Budgeting, and Reporting System

SPS: Standard Procurement System

TPDB: Trading Partner Database

UDF: User-Defined File

VAN: Value Added Network